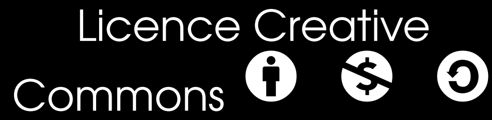
Informatique

MPSI 2010-2011



Une année d'informatique en MPSI

Guillaume CONNAN

Licence Creative Commons

TABLE DES MATIÈRES

0 Premiers pas **5**

0.1 Généralités . . . 6
- 0.1.1 Algorithme et programme . . . 6
- 0.1.2 Spécification . . . 7

0.2 Une galerie d'exemples . . . 8
- 0.2.1 Posons le problème . . . 8
- 0.2.2 Spécifions le problème . . . 8
- 0.2.3 Premier algorithme : fonction récursive . . . 8
- 0.2.4 Deuxième algorithme : boucle « pour » . . . 11
- 0.2.5 Troisième algorithme : boucle « tant que » . . . 12
- 0.2.6 Exponentiation logarithmique . . . 13

0.3 Quelques types prédéfinis . . . 13
- 0.3.1 Les entiers . . . 13
- 0.3.2 Les flottants . . . 14
- 0.3.3 Les caractères et chaînes de caractères . . . 14
- 0.3.4 Les booléens . . . 14

0.4 Les structures conditionnelles . . . 15

0.5 La programmation impérative : au plus près de la machine . . . 15
- 0.5.1 L'affectation . . . 15
- 0.5.2 Structures itératives . . . 16
- 0.5.3 Les listes : premier contact . . . 18

0.6 Au plus près de l'humain : la programmation récursive . . . 20
- 0.6.1 Qu'est-ce que c'est ? . . . 20
- 0.6.2 Deux manières d'aborder un même problème : les tours de Hanoï . . . 22
- 0.6.3 Un exemple déjà traité en impératif . . . 24

0.7 EXERCICES . . . 25
- 0.7.1 Généralités . . . 25
- 0.7.2 Structures conditionnelles . . . 25
- 0.7.3 Programmation impérative . . . 25
- 0.7.4 Récursion . . . 26

1 Méthodes de programmation **29**

1.1 PASCAL . . . 30
- 1.1.1 Types prédéfinis . . . 30
- 1.1.2 Priorité des opérateurs . . . 35
- 1.1.3 Structures conditionnelles . . . 36
- 1.1.4 Structures de répétition . . . 36
- 1.1.5 Types définis par l'utilisateur . . . 38
- 1.1.6 Récursion . . . 39

1.2 Correction et terminaison d'un algorithme . . . 40

1.2.1 Algorithmes impératifs 40
1.2.2 Algorithmes récursifs 41
1.3 Récursion mutuelle 43
1.4 Diviser pour régner 44
1.5 EXERCICES 45
1.6 DES SOLUTIONS 48

2 Complexité **54**
2.1 Premières définitions 55
2.2 Premier exemple : exponentiation 56
2.2.1 Méthode naïve 56
2.2.2 Diviser pour régner 57
2.3 Recherche dans un tableau 58
2.3.1 Préliminaires 58
2.3.2 Recherche séquentielle 58
2.3.3 Recherche dichotomique 60
2.4 Algorithmes de tri 62
2.4.1 Permutation 62
2.4.2 Tri par sélection 62
2.4.3 Tri par insertion 63
2.4.4 Tri à bulles 65
2.4.5 Tri fusion 69
2.4.6 Tri Shell 70
2.4.7 Tri rapide 70
2.5 EXERCICES 71
2.6 DES SOLUTIONS 74

3 Structures de données I **77**
3.1 Enregistrements 78
3.1.1 Une classe de rationnels 78
3.1.2 Méthode de HÉRON 79
3.1.3 Fractions continues 79
3.2 Pointeurs et pointés 80
3.3 Listes chaînées définies récursivement 81
3.3.1 Qu'est-ce que c'est ? 81
3.3.2 Comment ? 81
3.3.3 Outils de base 82
3.3.4 Fractions continues 85
3.4 Listes chaînées définies itérativement 86
3.4.1 Principe 86
3.4.2 Outils de base 86
3.5 Piles 86
3.5.1 Qu'est-ce que c'est ? 86
3.5.2 Outils 87
3.5.3 Évaluation d'une expression arithmétique postfixée 88
3.6 EXERCICES 91
3.7 DES SOLUTIONS 92

Guillaume Connan, Licence Creative Commons, 2010-2011

4 Logique des propositions 99
4.1 Généralités 100
4.2 Syntaxe 100
4.2.1 Les symboles 100
4.2.2 Démonstration par induction 101
4.2.3 Sous-formules 101
4.3 Sémantique 102
4.3.1 Valeurs et tables de vérité 102
4.3.2 Calcul propositionnel et Calcul dans $\mathbf{F_2}$ 102

CHAPITRE

0

Premiers pas

After more than a decade of intense research,
Derek unveils his calculation for the value of pi.

1 Généralités

1 1 Algorithme et programme

```
Fonction algorithme breton( masse m : entier: ): gâteau délicieux
beurre,sucre,farine,œuf: entier
beurre← m/4
sucre← m/4
farine← m/4
œuf← m/4
Début
  | Couper le beurre en petits moreaux
  |  Le mettre à fondre doucement au bain-marie de
préférence.
  | Dès qu'il est fondu arrêter.
  | Laisser refroidir mais attention : le beurre doit être
encore
  | liquide !
  |  Il ne doit pas redevenir solide.
  |  Mettre le four à préchauffer à 160 degrés (th 5).
  |  Mettre les œufs entiers avec le sucre dans un saladier.
  | Battre longuement le mélange pour qu'il blanchisse
  |  et devienne bien mousseux.
  |  Y ajouter le beurre fondu ET FROID.
  |  Rajouter progressivement à l'appareil la farine
  |  en l'incorporant bien.
  | Cela doit donner une pâte élastique et un peu épaisse.
  |  Verser la préparation dans un moule à manqué
  | ou à cake bien beurré.
  |  Laisser cuire environ une heure.
  | Il faut surveiller le gâteau régulièrement.
  | TantQue  Le dessus du gâteau n'est pas blond foncé Faire
  |   | Continuez la cuisson
  | FinTantQue
  | Si  Il semble brunir trop vite Alors
  |   |  il faut baisser un peu le four
  |   | Il faut mettre une feuille d'aluminium sur le dessus.
  | FinSi
  | Si lorsqu'une pique plantée en son milieu ressort sèche
Alors
  |   | le gâteau est cuit
  | Sinon
  |   | Continuez la cuisson.
  | FinSi
  | Retirez le gâteau
Fin
```

Un algorithme est une description rigoureuse en un nombre fini d'étapes de la résolution d'un problème.
La notion d'algorithme existe depuis bien avant l'existence des ordinateurs et même bien avant la naissance du mot lui-même puisqu'il est tiré du nom du célèbre mathématicien persan أبو عبد الله محمد بن موسى الخوارزمي ou si vous préférez Abu 'Abdallah Muhammad ibn Musa al-Khwarizmi qui vécut au IXe siècle de notre ère, c'est-à-dire près de 3000 ans après les premiers algorithmes de calcul connus mis au point en Mésopotamie.
Vous utilisez depuis longtemps des algorithmes, à la manière de M. Jourdain, sans toujours le savoir, par exemple lorsque vous lisez une recette de cuisine, lorsque vous recherchez un mot dans le dictionnaire, lorsque vous effectuez une addition posée.
Cependant, cette année, nous nous contenterons d'algorithmes destinés à être utilisés par un ordinateur. Nous introduirons donc un langage algorithmique le moins ambiguë possible et le plus standard afin de pouvoir mettre en œuvre ces algorithmes sur n'importe quelle machine.
Même si nos algorithmes doivent être indépendants de tout langage de programmation, il sera enrichissant de les tester « en vrai » sur machine. Nous serons donc amenés à parler à la machine. Pour cela nous utiliserons un langage servant d'interface entre la machine et nous car tout être humain normalement constitué ne peut décemment entamer une longue conversation avec un ordinateur en langage machine.
Au programme de MP, nous avons le choix entre PASCAL et CAML. C'est PASCAL qui a été choisi dans notre lycée. Cependant, lors de l'enseignement commun du premier trimestre, nous nous réservons le droit de proposer quelques algorithmes en Maple.

1 2 Spécification

Les deux phrases suivantes ont un sens pour nous [a] :

- *On doit porter un casque sur une moto.*
- *On doit porter les animaux sur les escaliers mécaniques.*

On utilise deux fois *on doit porter* mais le sens n'est pas le même :

- Il n'est bien sûr pas obligatoire d'avoir un chien dans les bras pour pouvoir utiliser un escalier mécanique. Il faudrait donc plutôt dire :« dans le cas où vous avez un animal, portez-le sur l'escalier mécanique. Si vous n'en avez pas, vous pouvez prendre l'escalier sans contrainte ».
- Si on n'a pas de casque sur soi, on ne peut pas s'en passer pour monter sur une moto, comme on peut se passer d'avoir un chien avec soi pour monter sur un escalier mécanique. Il faudrait plutôt dire :« il est obligatoire d'avoir toujours un casque pour pouvoir monter sur une moto. »

Il faudra donc être très précis et non ambigus dans notre formulation. C'est pourquoi nous préférerons un langage formel au langage naturel.

a. **?**, .

Enfin, spécifier, ce n'est pas dire comment sont effectués les calculs mais ce qu'ils calculent.

Exemple

Spécifions une fonction qui a pour données un nombre en base dix et qui a pour résultat son écriture en base deux.

Précisons un peu :

- La donnée est un nombre entier positif écrit en base dix (c'est la *précondition*).
- Le résultat est une liste d'entiers appartenant à $\{0;1\}$ dans l'ordre décroissant des puissances de deux associées. On convient de commencer par la plus grande puissance dont le coefficient est non nul (c'est la *postcondition*).

Par exemple, si on entre 11 on obtiendra $[1,0,1,1]$.
En effet, $11 = 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0$.
On ne sait pas encore *comment* calculer le résultat mais on sait déjà CE QUE CALCULE cette fonction et on peut donc l'intégrer dans des calculs plus complexes.

2 Une galerie d'exemples

2 1 Posons le problème

Avant d'être capable d'écrire des algorithmes corrects, observons quelques exemples afin de nous donner une idée de la tâche à accomplir.

Exemple

Étant donné un entier relatif a et un entier naturel n, on voudrait obtenir a^n qui sera un entier relatif.

2 2 Spécifions le problème

```
puissance(a:entier;n:entier positif):entier
puissance(a,n) renvoie a à la puissance n
```

2 3 Premier algorithme : fonction récursive

```
Fonction puissance( a : entier ; n : entier positif ) : entier
{ Ceci est un commentaire non évalué par la machine }
{ puissance(a,n) renvoie a à la puissance n }
{ C'est un algorithme récursif }
Début
  Si n=0 Alors
    1
  Sinon
    a*puissance(a,n-1)
  FinSi
Fin
```

Recherche

– Que fait cet algorithme ?
– Peut-on le prouver ?
– Comment caractériser son efficacité ?

Traduction possible en Pascal :

```
FUNCTION Puissance(a:INTEGER;n:INTEGER):INTEGER;
 BEGIN
  IF n=0
  THEN Puissance:=1
  ELSE Puissance:=a*Puissance(a,n-1);
 END;
```

Mais Pascal ne fonctionne pas en mode interactif. Il faut donc inclure cette fonction dans un programme.

```
PROGRAM Puissance_rec;
VAR a:INTEGER; n:INTEGER;

 FUNCTION Puissance(a:INTEGER;n:INTEGER):INTEGER;
  BEGIN
   IF n=0
   THEN Puissance:=1
   ELSE Puissance:=a*Puissance(a,n-1);
  END;

BEGIN {programme principal}
  WRITE('Entrer a : ');READLN(a);
  WRITE('Entrer n : ');READLN(n);
  WRITELN;
  WRITE(a,'^',n,' = ',Puissance(a,n));
  READLN;
END.
```

et cela donne :

Réponse du logiciel

```
Entrer a : 2
Entrer n : 5

2^5 = 32
```

Oui mais...On n'a pas obéit à notre spécification : le cas n négatif n'a pas été traité.

```
PROGRAM Puissance_rec;
VAR a:INTEGER; n:INTEGER;

 FUNCTION Puissance(a:INTEGER;n:INTEGER):INTEGER;
```

Guillaume Connan, Licence Creative Commons BY: $\circledS$ $\circledC$, 2010-2011

```
  BEGIN
   IF n=0
   THEN Puissance:=1
   ELSE Puissance:=a*Puissance(a,n-1);
  END;

BEGIN {programme principal}
  WRITE('Entrer a : ');READLN(a);
  WRITE('Entrer n : ');READLN(n);
  WRITELN;
  IF n<0
  THEN WRITE('L exposant doit etre positif')
  ELSE WRITE(a,'^',n,' = ',Puissance(a,n));
  READLN;
END.
```

Dans un autre langage comme Caml, l'idée est la même mais la manière de l'implanter et de l'utiliser diffèrent :

```
# exception Exposant_negatif;;


# let rec puissance(a,n)=
    if n<0
    then raise Exposant_negatif
    else
      if n==0
      then 1
      else a*puissance(a,n-1);;
```

Ou en Maple :

```
> puissance:=proc(a,n)
    if n<0 then RETURN(`exposant negatif !!`)
    elif n=0
    then 1
    else a*puissance(a,n-1)
    fi
  end:
```

Ou en Python :

```
>>> def puissance(a,n):
...     if n<0:
...         return("exposant negatif")
...     elif n==0:
...         return(1)
...     else:
...         return(a*puissance(a,n-1))
```

Ou en XCAS :

```
puissance(a,n):={
 si n<0 alors retourne("exposant negatif !!")
        sinon si n==0 alors retourne(1)
              sinon retourne(a*puissance(a,n-1))
              fsi
 fsi
}
```

2 4 Deuxième algorithme : boucle « pour »

```
Fonction puissance( a : entier ; n : entier positif ) : entier
{ puissance(a,n) renvoie a à la puissance n }
{ C'est un algorithme impératif utilisant une boucle pour }
Variable
  temp,k : entier
Début
  temp ← 1
  Pour k variantDe 1 à n Faire
    { On ne rentre dans la boucle que si k>=1 }
    temp ← temp*a
  FinPour
  Retourner temp
Fin
```

Recherche

- Que fait cet algorithme ?
- Qu'est-ce qui le différencie du précédent ?
- Peut-on le prouver ?
- Comment caractériser son efficacité ?

En Pascal :

```
PROGRAM Puissance_it;
VAR a:INTEGER; n:INTEGER;

FUNCTION Puissance(a:INTEGER;n:INTEGER):INTEGER;
VAR temp:INTEGER; k:INTEGER;
 BEGIN
  temp:=1;
  FOR k:=1 TO n DO temp:=temp*a;
  Puiss:=temp;
 END;


BEGIN {programme principal}
  WRITE('Entrer a : ');READLN(a);
```

```
  WRITE('Entrer n :  ');READLN(n);
  WRITELN;
  IF n<0
  THEN WRITE('L exposant doit etre positif')
  ELSE WRITE(a,'^',n,' = ',Puissance(a,n));
  READLN;
END.
```

On remarque que, en Pascal, que la fonction de calcul de la puissance soit récursive ou itérative, la manière de l'utiliser dans le programme principal est la même.

2 5 Troisième algorithme : boucle « tant que »

```
Fonction puissance( a : entier ; n : entier positif ) : entier
{ puissance(a,n) renvoie a à la puissance n }
{ C'est un algorithme impératif utilisant une boucle tant que }
Variable
 | temp,k : entier
Début
 | temp ← 1
 | k ← n
 | TantQue k>0 Faire
 |  | { On ne rentre dans la boucle que si k>0 }
 |  | k ← k-1
 |  | temp ← a*temp
 | FinTantQue
 | Retourner temp
Fin
```

En Pascal :

```
PROGRAM Puissance_tq;
VAR a:INTEGER; n:INTEGER;

FUNCTION Puissance(a:INTEGER;n:INTEGER):INTEGER;
VAR temp:INTEGER; k:INTEGER;
 BEGIN
  temp:=1;
  k:=n;
  WHILE k>0  DO
    BEGIN
    k:=k-1;
    temp:=temp*a;
    END;
  Puiss:=temp;
 END;

BEGIN {programme principal habituel}
END.
```

2 6 Exponentiation logarithmique

Le principe est simple :

$$\begin{cases} a^0 & = & 1 \\ a^n & = & \left(a^2\right)^{\frac{n}{2}} & \text{, si } n \text{ est pair et } n > 0 \\ a^n & = & a\left(a^2\right)^{\frac{n-1}{2}} & \text{, si } n \text{ est impair} \end{cases}$$

Recherche Écrivez un algorithme utilisant ces relations de récurrence. Qu'y gagne-t-on ?

3 Quelques types prédéfinis

Les variables, les constantes, les arguments introduits dans les algorithmes peuvent être de types différents. Tous les langages ont des types prédéfinis à peu près communs.

3 1 Les entiers

Un entier naturel est un entier positif ou nul. Le choix à faire (c'est-à-dire le nombre de bits[b] à utiliser) dépend de la fourchette des nombres que l'on désire utiliser. Pour coder des nombres entiers naturels compris entre 0 et 255, il nous suffira de 8 bits (un octet) car $2^8 = 256$. D'une manière générale un codage sur n bits pourra permettre de représenter des nombres entiers naturels compris entre 0 et 2^{n-1}. Les processeurs disponibles actuellement proposent des codages sur 32 ou 64 bits. Cependant, les logiciels ne tiennent pas toujours compte des 64 bits et vous avez beau disposer d'un processeur 64 bits, il sera utilisé comme s'il était de type 32 bits (notamment si le système d'exploitation est microsoft-windows...).
Pour représenter un nombre entier naturel après avoir défini le nombre de bits sur lequel on le code, il suffit de ranger chaque bit dans la cellule binaire correspondant à son poids binaire de la droite vers la gauche, puis on « remplit » les bits non utilisés par des zéros.

Pour les entiers relatifs, le bit situé à l'extrême gauche (le bit de poids fort), représentera le signe : 0 pour un nombre positif, 1 pour un nombre négatif. Cela restreint donc l'étendue des nombres disponibles.
Un entier relatif positif est donc codé comme un entier naturel mais avec un 0 à gauche.
Pour les entiers négatifs, on utilise une petite ruse : le complément à 2 que nous étudierons à titre d'exercice.
Ainsi, sur les ordinateurs qui sont à notre disposition, tout introduction d'un entier dépassant $2^{31} - 1 = 2\,147\,483\,647$ engendrera une erreur.

b. Le mot « bit » est la contraction des mots anglais binary digit, qui signifient « chiffre binaire », avec un jeu de mot sur bit, « morceau ». Il ne faut pas confondre avec le mot « byte », en français « multiplet », qui désigne un assemblage de bits, en général 8 (un octet).

Guillaume Connan, Licence Creative Commons BY: $\textcircled{\$}$ $\textcircled{=}$, 2010-2011

3 2 Les flottants

La norme IEEE définit la façon de coder un nombre réel. Cette norme se propose de coder le nombre sur 32 bits et définit trois composantes :

- le signe est représenté par un seul bit, le bit de poids fort (celui le plus à gauche) ;
- l'exposant est codé sur les 8 bits consécutifs au signe ;
- la mantisse, c'est-à-dire le nombre « sans la virgule », sur les 23 bits restants.

La précision des nombres réels est approchée. Elle dépend du nombre de positions décimales, suivant le type de réel elle sera au moins :

- de 6 chiffres après la virgule pour un système 32 bits ;
- de 15 chiffres après la virgule pour un système 64 bits.

Ces limitations peuvent causer quelques surprises. Par exemple, vous pouvez tester sur Maple :

```
> 100000+0.0000002;
```

Réponse du logiciel

```
100000
```

Remarque

Ces problèmes d'implémentation des entiers et des flottants ne concernent pas l'algorithmique mais la programmation dans un certain langage, sur un certain système. On n'en tiendra donc pas forcément compte.

3 3 Les caractères et chaînes de caractères

Le type « caractère » permet de stocker le code d'un caractère, c'est-à-dire un nombre entier. Une chaîne de caractères est une suite ordonnée de caractères. La représentation d'une chaîne de caractères dépend d'un système à un autre.

3 4 Les booléens

Avant d'étudier les structures conditionnelles, il nous faut nous habituer aux tests, aux booléens et à leurs opérateurs, qui ne sont pas d'un usage aussi naturel que les types flottants et entiers.

- Une variable de type booléen n'a que deux valeurs possibles : VRAI ou FAUX.
- Ces booléens sont en particulier les valeurs retournées par les opérateurs de comparaison : `=`, `<`, `<=`, `>`, `>=`, `<>`.
- Ils ont en général 4 opérateurs : **ET**, **OU**, **OUEX**, **NON**.

Guillaume Connan, Licence Creative Commons BY: \$ C, 2010-2011

Recherche

Par exemple, quelle sera la réponse à :

- $3 > 2$ ET $2 > 2$?
- $3 > 2$ ET $2 \geqslant 2$?
- $3 > 2$ OU $2 > 2$?
- $3 > 2$ OU $2 \geqslant 2$?
- $3 > 2$ OUEX $2 \geqslant 2$?
- $3 > 2$ OUEX $2 > 2$?
- NON $2 > 2$?
- NON $3 > 2$?

4 Les structures conditionnelles

Elles sont communes aux deux types de programmation que nous étudierons. C'est le fameux SI...ALORS...SINON...
Voici un bel algorithme :

```
Fonction zorg( a,b,c : entier ) : caractère
Début
  Si a<>b Alors
    Si a+1=b Alors
      Retourner 'aarrgh'
    Sinon
      Si c<>7 Alors
        Retourner 'glurp'
      Sinon
        Retourner 'pas glop'
      FinSi
    FinSi
  Sinon
    Retourner 'eviv bulgroz'
Fin
```

Recherche

Que répond

- `zorg(4,5,7)` ?
- `zorg(4,5,8)` ?
- `zorg(4,6,7)` ?

5 La programmation impérative : au plus près de la machine

5 1 L'affectation

L'instruction représentée par :

```
a ← 2
```

qui se lit « a reçoit 2 » réserve (affecte) une zone de la mémoire de l'ordinateur qui portera l'étiquette a et qui contiendra 2.

Recherche

Donnez le contenu de a et b après la suite d'affectations suivante :

```
a ← 1
b ← 2
a ← a+b
b ← a-b
a ← a-b
```

Un algorithme impératif va donc modifier chronologiquement l'état de la mémoire. Les valeurs des variables vont sans cesse évoluer. Cependant, la plupart du temps, les notations des structures itératives utilisées n'expriment pas clairement l'état des variables. Il faudra donc, pour bien maîtriser l'algorithme, le commenter pour décrire les états parcourus par le système.

Danger

On veillera à ne pas modifier les arguments d'une fonction avec quelque chose du style :

```
Fonction non(a,b,c:entier):entier
a←a+1
```

Il faut introduire une variable locale prenant au départ la valeur de l'argument puis la modifier :

```
Fonction oui(a,b,c:entier):entier
Variable
  | temp:entier
temp←a
temp←temp+1
```

5 2 Structures itératives

5 2 1 Tant que

Exemple

Soit deux entiers naturels n et N tels que $n \leqslant N$. On voudrait calculer la somme des entiers de n à N.

Les notations mathématiques peuvent ici nous guider. En effet, cette somme $S(n,N)$ s'écrit :

$$S(n,N) = \sum_{k=n}^{k=N} k = n + (n+1) + (n+2) + \cdots + (N-1) + N$$

```
Fonction somme(n,N:entier):entier
{ On doit avoir n<=N. On obtient la somme des entiers successifs
de n à N compris }
{ Nous allons créer une variable provisoire qui contiendra la
somme en construction ainsi que l'entier provisoire qui va être
itéré. }
Variable
Stemp,ktemp:
Début
  Stemp← 0
  ktemp ← n
  { On choisit d'initialiser la somme à 0 et l'indice d'itération
à n. }
  TantQue ktemp <= N Faire
    { On entre dans la boucle jusqu'à la valeur N comprise }
    Stemp← Stemp+ktemp
    { À l'état initial on commence donc par rajouter n à 0 }
    ktemp← ktemp+1
```

```
    { ktemp est incrémenté après la somme. }
    {  À la dernière étape, ktemp vaut N, }
    {  est rajouté à Stemp puis ktemp reçoit N+1 }
    {  et on sort donc de la boucle. }
    {  Dans l'état final on a donc la somme des entiers de n à
N. }
  FinTantQue
  Retourner Stemp
Fin
```

Avec Maple :

```
> Somme:=proc(n,N)
   local Stemp,ktemp;
   Stemp:=0;
   ktemp:=n;

   while ktemp<=N do
     Stemp:=Stemp+ktemp;
     ktemp:=ktemp+1;
   od;

   RETURN(Stemp)
  end:
```

Vérifions que notre algorithme fait bien ce qu'on attend de lui.
La propriété $\mathscr{P}$(**ktemp**) : « **Stemp**= S(n,**ktemp**) et **ktemp**$\leqslant$ N » est l'*invariant de boucle*.
Au départ, on a bien $\mathscr{P}(n)$: **Stemp**= $\mathrm{S}(n,n) = n$ et $n \leqslant \mathrm{N}$.
À chaque fin d'occurrence de la boucle, cette propriété reste vraie ainsi qu'en sortie.

Guillaume Connan, Licence Creative Commons BY: $ C , 2010-2011

Recherche

On modifie légèrement l'ordre des instructions de cet algorithme :

```
ktemp← ktemp+1
Stemp← Stemp+ktemp
```

Y a-t-il d'autres choses à modifier ?

5 2 2 Pour

En anglais, c'est « for » et sur MAPLE c'est :

```
for var from debut to fin by increment do
    instructions;
od;
```

Par exemple :

```
> Somme:=proc(n::posint)
  local S,k;
  S:=0;
   for k from 1 to n by 1 do
     S:=S+k;
   od;
 RETURN(S)
 end:
```

On peut arrêter l'itération dès qu'une condition a été réalisée (et se passer du `to`) :

```
> Somme:=proc(n::posint)
  local P,k;
  S:=0;
   for k from 1 while P<=n do
     S:=S+k;
   od;
 RETURN(S)
 end:
```

5 3 Les listes : premier contact

5 3 1 Généralités

Une liste est un objet très important en informatique. Nous l'étudierons plus précisément après décembre. Sachons juste pour l'instant qu'il s'agit d'un ensemble ordonné d'objets quelconques qui peuvent d'ailleurs être des listes.
On les note habituellement à l'aide de crochets :

```
L←[a,b,1,2,"maman",x->cos(x)]
```

On peut accéder à un élément de la liste par son numéro (on commence généralement à compter à partir de 0). Ainsi, `L[0]` est `a`, `L[3]` est `2`.
On peut extraire une sous-liste : `L[1..3]` donnera `[b,1,2]`.

On peut *concaténer* des listes, c'est-à-dire les « coller ». Nous utiliserons conventionnellement pour représenter cet opérateur le symbole **+**.

5 3 2 Avec MAPLE

Les listes sont un type de variable particulier que peut manipuler MAPLE. Une liste se présente sous la forme d'une suite d'objets séparés par des virgules entre crochets :

```
> L:=[1,2,a,g,maman,ln(5),3/2,x->3*x+2,[7,9]]
```

Cette suite est ORDONNÉE. Ses éléments sont donc numérotés. On peut en extraire un élément particulier :

```
> L[5]
```

ou bien tous les éléments de la liste :

```
op(L)
```

ou bien une sous-liste :

```
> L[3..6];
```

On peut connaître la taille d'une liste :

```
> nops(L);
```

On peut ajouter un élément à une liste :

```
> L:=[op(L),nouveau]
```

On peut *concaténer* deux listes :

```
> L1:=[1,2,3,4];
> L2:=[a,b,c];
> L:=[op(L1),op(L2)]
```

La liste vide se note :

```
> V:=[];
> nops(V);
```

On peut effectuer des opérations sur une liste car c'est un type de variable comme un autre :

```
> S:=L1+[op(L2),d];
> P:=3*L1;
```

Guillaume Connan, Licence Creative Commons, 2010-2011

6 Au plus près de l'humain : la programmation récursive

6 1 Qu'est-ce que c'est ?

La récursion est un mécanisme puissant permettant, comme l'itération, d'exprimer la répétition des opérations, mais de manière plus concise et en ayant la possibilité de se passer d'affectation et ainsi éviter les problèmes de surveillance de l'état de la mémoire.
Il ne se limite pas au cas des suites numériques définies par une relation de récurrence.
Par exemple, définissons un caractère comme un élément de l'ensemble constitué des 26 lettres de l'alphabet latin : $\left\{ 'a', 'b', \cdots, 'z' \right\}$
On peut donner une définition récursive d'un mot :
Un mot est soit un caractère, soit un caractère suivi d'un mot.
Avec cette définition, « papa » est un mot mais « père » ne l'est pas.

6 1 1 Une récursion concrète

On place un certain nombre d'élèves en les classant par ordre décroissant de taille. Le plus grand voudrait connaître sa propre taille mais chaque élève ne peut mesurer que son écart de taille avec son voisin. Seul le plus petit connaît sa taille.
Le plus grand mesure son écart avec son voisin et garde le nombre en mémoire. Le voisin fait de même avec son propre voisin et ainsi de suite. Enfin l'avant dernier fait de même puis le plus petit lui transmet sa taille : il peut donc calculer sa propre taille et transmet la au suivant et ainsi de suite : chacun va ainsi pouvoir calculer sa taille de proche en proche....

6 1 2 Analyse rapide e d'un algorithme récursif

Il n'est pas trop compliqué de comprendre que l'entier suivant n est égal à 1+ l'entier suivant $n-1$.
Maintenant, il faudrait connaître « la taille du plus petit », c'est-à-dire une *condition terminale*. Si on ne s'intéresse qu'aux entiers naturels, le plus petit sera 0 et on fixe que le successeur de 0 est 1.
On peut alors écrire un algorithme ainsi :

```
Fonction successeur(n : entier naturel) : entier naturel
Début
  Si n=0 Alors
    Retourner 1
  Sinon
    Retourner 1+successeur(n-1)
  FinSi
Fin
```

Par exemple sur Maple :

```
> successeur:=proc(n)
    if n=0 then RETURN(1)
    else RETURN(1+successeur(n-1))
    fi
```

```
end:
```

Que se passe-t-il dans le cœur de l'ordinateur lorsqu'on entre **successeur(3)** ?

- On entre comme argument 3 ;
- Comme 3 n'est pas égal à 0, alors**successeur(3)** est stocké en mémoire et vaut

  1+successeur(2) ;
- Comme 2 n'est pas égal à 0, alors**successeur(2)** est stocké en mémoire et vaut

  1+successeur(1) ;
- Comme 1 n'est pas égal à 0, alors**successeur(1)** est stocké en mémoire et vaut

  1+successeur(0) ;
- Cette fois,**successeur(0)** est connu et vaut 1 ;
- **successeur(1)** est maintenant connu et vaut **1+successeur(0)** c'est-à-dire 2 ;
- **successeur(2)** est maintenant connu et vaut **1+successeur(1)** c'est-à-dire 3 ;
- **successeur(3)** est maintenant connu et vaut **1+successeur(2)** c'est-à-dire 4 : c'est fini !

C'est assez long comme cheminement mais ce n'est pas grave car *c'est l'ordinateur qui effectue le « sale boulot »* ! Il stocke les résultats intermédiaires dans une *pile* et n'affiche finalement que le résultat.
Comme il calcule vite, ce n'est pas grave. L'élève ou le professeur ne s'est occupé que de la définition récursive (mathématique !) du problème.
Maple calcule facilement**successeur(543)** mais ensuite on dépasse les possibilités de calcul du logiciel.
Et oui, un langage impératif ne traite pas efficacement le problème de pile, c'est pourquoi pendant longtemps seuls les algorithmes impératifs ont prévalus.
CAML est un langage fonctionnel et gère très efficacement la mémoire de l'ordinateur pour éviter sa saturation lors d'appels récursifs.
Le programme s'écrit comme en mathématique (mais en anglais...) :

```
# let rec successeur(k)=
  if k=0 then 1
  else 1+successeur(k-1);;
```

Alors par exemple :

```
# successeur(36000);;
- : int = 36001
```

Mais

```
# successeur(3600000);;
Stack overflow during evaluation (looping recursion?).
```

La *pile* où sont stockés les résultats intermédiaires créés par la récursion est en effet saturée.
Aujourd'hui, les langages fonctionnels sont très efficaces et gèrent de manière intelligente la *pile*. Ils le font soit automatiquement, soit si la fonction utilise une *récursion terminale*, c'est-à-dire si l'appel récursif à la fonction n'est pas *enrobé* dans une autre fonction.
Ici, ce n'est pas le cas car l'appel récursif **`successeur(k-1)`** est enrobé dans la fonction $x \mapsto 1+x$.
On peut y remédier en introduisant une fonction intermédiaire qui sera récursive terminale :

```
# let rec successeur_temp(k,resultat)=
        if k=0 then resultat
        else successeur_temp(k-1,resultat+1);;
```

puis on appelle cette fonction en prenant comme résultat de départ 1 :

```
# let successeur_bis(k)=
  successeur_temp(k,1);;
```

Alors :

```
# successeur_bis(360000000);;
- : int = 360000001
```

Il n'y a donc aucun problème pour traiter 360 millions d'appels récursifs !

6 2 Deux manières d'aborder un même problème : les tours de Hanoï

6 2 1 Le principe

Ce casse-tête a été posé par le mathématicien français Édouard LUCAS en 1883.
Le jeu consiste en une plaquette de bois où sont plantés trois piquets. Au début du jeu, n disques de diamètres croissant de bas en haut sont placés sur le piquet de gauche. Le but du jeu est de mettre ces disques dans le même ordre sur le piquet de droite en respectant les règles suivantes :

- on ne déplace qu'un disque à la fois ;
- on ne peut poser un disque que sur un disque de diamètre supérieur.

Essayez avec 2 puis 3 disques... Nous n'osons pas vous demander de résoudre le problème avec 4 disques !

6 2 2 Non pas comment mais pourquoi : version récursive

En réfléchissant récursivement, c'est enfantin !... Pour définir un algorithme récursif, il nous faut régler un cas simple et pouvoir simplifier un cas compliqué.

- *cas simple* : s'il y a zéro disque, il n'y a rien à faire !
- *simplification d'un cas compliqué* : nous avons n disques au départ sur le premier disque et nous savons résoudre le problème pour $n-1$ disques. Appelons A, B et C les piquets de gauche à droite. Il suffit de déplacer les $n-1$ disques supérieurs de A vers B (hypothèse de récurrence) puis on déplace le plus gros disque resté sur A en C. Il ne reste plus qu'à déplacer vers C les $n-1$ disques qui étaient en B. (hypothèse de récurrence).

On voit bien la récursion : le problème à n disques est résolu si on sait résoudre le cas à $n-1$ disques et on sait quoi faire quand il n'y a plus de disques.
On sait pourquoi cet algorithme va réussir mais on ne sait pas comment s'y prendre étape par étape.
On va donc appeler à l'aide l'ordinateur en lui demandant d'écrire les mouvements effectués à chaque étape.
On commence par créer une fonction qui affichera le mouvement effectué :

```
let mvt depart arrivee=
print_string
("Deplace un disque de la tige "^depart^" vers la tige "^arrivee)
    ;
print_newline();;
```

Listing 1 – mouvement élémentaire

Le programme proprement dit :

```
let rec hanoi a b c= function
    | 0 -> ()  (*0 disque : on ne fait rien*)
    | n -> hanoi a c b (n-1); (*n-1 disques sont places dans l'
        ordre de a vers b*)
           mvt a c; (*on deplace le disque restant en a vers c*)
           hanoi b a c (n-1) (*n-1 disques sont places dans l'
              ordre de b vers c*);;
```

Listing 2 – résolution récursive du problème des tours de Hanoï

Les phrases entre (* et*) sont des commentaires.
Par exemple, dans le cas de 3 disques :

```
# hanoi "A" "B" "C" 3;;
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige A vers la tige B
Deplace un disque de la tige C vers la tige B
Deplace un disque de la tige A vers la tige C
Deplace un disque de la tige B vers la tige A
Deplace un disque de la tige B vers la tige C
Deplace un disque de la tige A vers la tige C
- : unit = ()
```

Guillaume Connan, Licence Creative Commons , 2010-2011

6 2 3 Non pas pourquoi mais comment : version impérative

Prenez un papier et un crayon. Dessinez trois piquets A, C et B et les trois disques dans leur position initiale sur le plot A.
Déplacez alors les disques selon la méthode suivante :

– déplacez le plus petit disque vers le plot suivant selon une permutation circulaire A-C-B-A ;
– un seul des deux autres disques est déplaçable vers un seul piquet possible.

Réitérez (en informatique à l'aide de boucles...) ce mécanisme jusqu'à ce que tous les disques soient sur C dans la bonne position.
On voit donc bien ici comment ça marche ; il est plus délicat de savoir pourquoi on arrivera au résultat.
Le programme est lui-même assez compliqué à écrire : vous trouverez une version en C (235 lignes...) à cette adresse :

http ://files.codes-sources.com/fichier.aspx?id=38936&f=HANOI%5cHANOI.C

6 3 Un exemple déjà traité en impératif

On définit la partie entière d'un réel x comme étant le plus grand entier inférieur à x.
On part du fait que la partie entière d'un nombre appartenant à $[0;1[$ est nulle.
Ensuite, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montrant que $\lfloor x \rfloor = 1 + \lfloor x-1 \rfloor$.
Si le nombre est négatif, on « monte » vers 0 en montrant que $\lfloor x \rfloor = -1 + \lfloor x+1 \rfloor$.

```
Fonction partie_entiere( x : flottant ) : entier
Début
    Si x>=0 et x<1 Alors
        Retourner 0
    Sinon
        Si x>=0 Alors
            Retourner 1+partie_entiere(x-1)
        Sinon
            Retourner -1+partie_entiere(x+1)
        FinSi
    FinSi
Fin
```

Avec Maple :

```
> partie_entiere:=proc(x)
    if x>=0 and x<1 then RETURN(0)
    elif x>0 then RETURN(1+partie_entiere(x-1))
    else RETURN(-1+partie_entiere(x+1))
    fi
  end:
```

On notera que l'algorithme récursif s'applique ici à des **réels** et non plus à des **entiers** : on dépasse ainsi le cadre des suites. Pour que l'algorithme fonctionne, on raisonne sur des intervalles et non plus sur une valeur initiale.

EXERCICES

Généralités

0 - 1 Type booléen

Sans utiliser de structure conditionnelle, écrivez un algorithme, qui, lorsqu'on lui fournit les trois coefficients d'un trinôme du second degré, renvoie VRAI si ce trinôme admet 2 racines distinctes et FAUX sinon.

0 - 2 Type booléen

Donnez un algorithme qui, deux nombres étant donnés, affiche VRAI s'ils sont de même signe et FAUX sinon.

0 - 3 Flottants et processeur

Sur Maple ou XCAS, observez et commentez ce que donne :

```
> x:=10; y:=-5; z:=5.00000001;
> (x*y)+(x*z);
> x*(y+z);
```

Structures conditionnelles

0 - 4

Écrivez un algorithme qui renvoie la valeur absolue d'un réel x.

0 - 5

Soit un système $\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$.
Déterminez un algorithme qui renvoie la solution de ce système si elle est unique.

0 - 6 Feu rouge

Créez un algorithme qui renvoie le successeur de la couleur d'un feu tricolore.

0 - 7 Mention au Bac

Créez un algorithme qui à une note de Bac renvoie la mention correspondante.

0 - 8 Remise

On entre un montant hors taxe ht. On effectue une remise sur ht selon la régle suivante :

- si $ht < 2500$€ alors il n'y a pas de remise ;
- si $2500 \leqslant ht < 4000$€ alors la remise est de 5% ;
- dans les autres cas la remise est de 8%.

La TVA est de 19,6%. On demande le prix TTC connaissant le prix HT.

0 - 9 Années bissextiles

Une année bissextile est une année dont le millésime, supérieur à 1583, est divisible par 4, sauf les limites de siècles qui ne sont pas multiples de 400. Ainsi 2 000 et 2 008 sont bissextiles mais 2 100 et 2 011 ne le sont pas.
Définissez alors un algorithme qui reconnaît si un millésime est celui d'une année bissextile. Vous pourrez utiliser les fonctions `quotient` et `reste` de la division euclidienne.

Programmation impérative

0 - 10

Écrire un algorithme qui calcule très rapidement x^{16} en n'utilisant qu'une multiplication par ligne et des affectations.

0 - 11

Déterminez un algorithme qui reçoit un nombre entier de secondes et qui renvoie quatre entiers correspondant à sa conversion en jours, heures, minutes et secondes. On pourra utiliser une fonction « quo » qui renvoie le quotient entier de la division de deux entiers.

0 - 12 Partie entière

Donnez un algorithme impératif renvoyant la partie entière d'un réel x.

Guillaume Connan, Licence Creative Commons BY: $\textcircled{\$}$ $\textcircled{=}$, 2010-2011

0 - 13 Division euclidienne

Déterminer un algorithme impératif qui renvoie le quotient de deux entiers naturels non nuls a et b.
Déterminer un algorithme qui renvoie le reste entier de a et b.

0 - 14 Intégration numérique

Déterminez un algorithme impératif qui donne une valeur approchée de $\int_{t=a}^{t=b} f(t)\,\mathrm{d}t$ pour une certaine fonction f continue sur un intervalle $[a;b]$ par la méthode des rectangles puis celle des trapèzes.

0 - 15

On veut étudier la suite de terme général $u_n = \frac{1}{n^2}$ ainsi que la suite $v_n = \sum_{k=1}^{n} u_k$, avec $n \in \mathbb{N}^*$.
Proposez un algorithme qui affiche u_n et v_n pour une valeur de n donnée.

Récursion

0 - 16 Factorielle

Écrivez un algorithme récursif qui donne $n!$ pour tout entier naturel n.

0 - 17 Somme

Écrivez un algorithme récursif qui calcule $\sum_{k=0}^{n} f(k)$ pour une fonction f donnée.

0 - 18 Division euclidienne

Déterminer un algorithme récursif qui renvoie le quotient de deux entiers naturels non nuls a et b.
Déterminer un algorithme récursif qui renvoie le reste entier de a et b.

0 - 19 Somme de chiffres

Donnez un algorithme récursif qui calcule la somme des chiffres d'un entier naturel n écrit en base 10.

0 - 20 Héron

HÉRON d'Alexandrie a trouvé une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif vingt siècles avant l'apparition des ordinateurs.
Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.
La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.
On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).
On obtient naturellement un algorithme... Donnez une version récursive prenant en argument le nombre dont on cherche la racine carrée, une première approximation x_0 et une précision `eps`.

0 - 21 Décimales de e

Posons $e_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$
Alors on a aussi

$$e_n = 1 + 1 + \frac{1}{2}\left(1 + \frac{1}{3}\left(1 + \frac{1}{4}\left(\cdots\left(\frac{1}{n-1}\left(1 + \frac{1}{n}\right)\right)\right)\right)\right)$$

Un exercice classique montre que $e - e_n \leqslant \frac{n+2}{n+1}\frac{1}{(n+1)!}$.
Ainsi, pour $n = 167$, on obtiendra 300 bonnes décimales au moins : trouvez-les !

0 - 22 Fractions continues dans Q

- Vous connaissez l'algorithme suivant :

$$\begin{aligned}
172 &= 3 \times 51 + 19 \quad (1)\\
51 &= 2 \times 19 + 13 \quad (2)\\
19 &= 1 \times 13 + 6 \quad (3)\\
13 &= 2 \times 6 + 1 \quad (4)\\
6 &= 6 \times 1 + 0 \quad (5)
\end{aligned}$$

- On peut donc facilement compléter la suite d'égalité suivante :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = \ldots$$

Guillaume Connan, Licence Creative Commons BY: $ \ominus $ ©, 2010-2011

– Quand tous les numérateurs sont égaux à 1, on dit qu'on a développé $\frac{172}{51}$ en fraction continue et pour simplifier l'écriture on note :

$$\frac{172}{51} = [3;2;\dots]$$

– Par exemple, on peut développer $\frac{453}{54}$ en fraction continue.

– Dans l'autre sens on peut écrire $[2;5;4]$ sous la forme d'une fraction irréductible.

On suppose connus les algorithmes donnant le reste et le quotient d'une division euclidienne.
Écrivez un algorithme donnant le développement en fraction continue d'un rationnel.
Écrivez ensuite un algorithme récursif qui effectue la transformation inverse.

0 - 23 Fractions continues dans R

En fait, l'étude précédente correspond à un cas particulier d'une définition plus générale.
Soit x un réel non entier. On construit une suite entière (z_n) de la manière suivante :

$$x = z_0 + \frac{1}{x_1} \qquad z_0 = \lfloor x \rfloor \qquad x_1 = \frac{1}{x - z_0}$$

puis, tant que x_k n'est pas entier :

$$x_k = z_k + \frac{1}{x_{k+1}} \qquad z_k = \lfloor x_k \rfloor \qquad x_{k+1} = \frac{1}{x_k - z_k}$$

Le développement de x en fractions continues est alors donné par $[z_0, z_1, z_2, \dots]$, cette liste pouvant être infinie.
Écrivez un algorithme récursif qui donne n chiffres du développement en fraction continue d'un nombre x donné.

0 - 24 Dichotomie

Pour résoudre une équation du type $f(x) = 0$, on recherche graphiquement un intervalle $[a, b]$ où la fonction semble changer de signe.
On note ensuite m le milieu du segment $[a, b]$. On évalue le signe de $f(m)$.
Si c'est le même que celui de $f(a)$ on remplace a par m et on recommence. Sinon, c'est b qu'on remplace et on recommence jusqu'à obtenir la précision voulue.
Écrivez un algorithme récursif qui prend comme argument une fonction f, les bornes a et b d'un intervalle et une précision eps.
On remarquera ici que la récursion se fait sans faire intervenir d'entier : le test d'arrêt est effectué sur la précision du calcul.

0 - 25

On définit deux fonction **`tete`** et **`queue`** qui renvoient respectivement le premier élément d'une liste et la liste privée de sa tête.
La syntaxe dépend des langages.
Sur Maple cela donne :

```
> tete:=proc(L)
   RETURN(L[1])
  end:
```

```
> queue:=proc(L)
   if nops(L)=1 then RETURN([])
   else RETURN(L[2..nops(L)])
   fi
  end:
```

Écrivez à présent une fonction récursive qui renvoie le maximum d'une liste de nombres.

0 - 26 Palindrome

Écrivez un algorithme récursif qui vérifie si une liste est un palindrome.

0 - 27 Nombres parfaits

Un nombre parfait est un nombre entier n strictement supérieur à 1 qui est égal à la somme de ses diviseurs (sauf n bien sûr !).

1. Il y en a un caché entre 1 et 10 : trouvez-le...
2. Écrivez un algorithme récursif qui donne la liste des diviseurs d'un entier autres que lui-même.
3. Écrivez un algorithme qui calcule la somme des éléments d'une liste.
4. Écrivez un algorithme qui détermine si un nombre entier est parfait.

5. Écrivez un algorithme récursif qui donne la liste des nombres parfaits entre 1 et un nombre n donné.

0 - 28 Décomposition en base 2

Une méthode pour obtenir l'écriture en base 2 d'un nombre est d'effectuer des divisions successives. Par exemple pour 11 :

$$\begin{array}{r|l} 11 & 2 \\ \hline 1 & 5 \end{array} \qquad \begin{array}{r|l} 5 & 2 \\ \hline 1 & 2 \end{array} \qquad \begin{array}{r|l} 2 & 2 \\ \hline 0 & 1 \end{array} \qquad \begin{array}{r|l} 1 & 2 \\ \hline 1 & 0 \end{array}$$

$$\begin{aligned} 11 &= \left(25+1\right) \\ &= \left(2\left(22+1\right)+1\right) \\ &= \left(2\left(2(21)+1\right)+1\right) \\ &= \left(2(2^2+1)+1\right) \\ &= 2^3+2+1 \\ &= 12^3+02^2+12^1+12^0 \end{aligned}$$

L'écriture de 11 en base 2 est donc 1011 : c'est la liste des restes obtenus mais dans l'ordre inverse.
La méthode est facilement généralisable.
Écrivez un algorithme récursif qui donne la décomposition d'un entier en base 2 sous forme d'une liste.

0 - 29 Algorithme de Hörner

Prenons l'exemple de $\mathrm{P}(x) = 3x^5-2x^4+7x^3+2x^2+5x-3$. Le calcul classique nécessite 5 additions et 15 multiplications.
On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} \mathrm{P}(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x + a_1}_{\text{on met } x \text{ en facteur}} \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)\, x + a_0 \end{aligned}$$

Ici cela donne $\mathrm{P}(x) = (((((3x)-2)x+7)x+2)x+5)x-3$ c'est-à-dire 5 multiplications et 5 additions.
En fait il y a au maximum 2degré de P opérations (voire moins avec les zéros).
On définit un polynôme par la donnée de ses coefficients dans l'ordre décroissant des exposants des monômes.
Déterminez un algorithme récursif qui prend comme argument les coefficients d'un polynôme P et un nombre x et qui calcule $\mathrm{P}(x)$ suivant le schéma de Hörner.

OPTION

1

Méthodes de programmation

1 PASCAL

Dans toute cette section, pour plus de détails, on pourra se référer à l'ouvrage de Claude DELANNOY [a].

1 1 Types prédéfinis

1 1 1 Les entiers

1 1 1 a INTEGER

Les opérateurs associés sont **+**, **-**, *****, **DIV**, **MOD**, **ABS**, **SQR**, **ODD**.

```
PROGRAM entier;
VAR i :  INTEGER;

BEGIN
   WRITE('Entrez un entier : ');
   READLN(i);
   WRITELN('Le double de ',i,' est ',2*i);
   READLN;
END.
```

Essayez avec $2^{30}-1$, $2^{31}-1$, 2^{31},...
INTEGER correspond aux entiers relatifs et **CARDINAL** aux entiers naturels.

1 1 1 b Les autres types d'entiers

En fait, **INTEGER** et **CARDINAL** dépendent de l'ordinateur utilisé. Pour pallier à cette incertitude, il existe des types d'entiers aux tailles fixées :

- **BYTE** : les entiers de 0 à $2^8-1=255$;
- **WORD** : les entiers de 0 à $2^{16}-1=65\,535$;
- **LONGWORD** : les entiers de 0 à $2^{32}-1=4\,294\,967\,295$;
- **QWORD** : les entiers de 0 à $2^{64}-1$;
- **SHORTINT** : les entiers de -128 à 127 ;
- **SMALLINT** : les entiers de $-32\,768$ à 32 767 ;
- **LONGINT** : les entiers de -2^{31} à $2^{31}-1$;
- **INT64** : les entiers de -2^{63} à $2^{63}-1$;

1 1 2 Les réels

1 1 2 a REAL

Les opérateurs associés sont **+**, **-**, *****, **/**, **ABS**, **SQR**, **SQRT**, **SIN**, **COS**, **ARCTAN**, **LN**, **EXP**, **FRAC**, **INT**, **ROUND**, **TRUNC**, **PI**.
Commentez ce programme :

```
PROGRAM reel;
CONST x = 1.0e20;
```

a. ?, .

```
     y = 2.0;

VAR adda : REAL;
    addb : REAL;

BEGIN
  adda:=(x+y)-x;
  addb:=(x-x)+y;
  WRITELN(adda:2:1,' = ',addb:2:1);
  READLN;
END.
```

1 1 2 b Les autres types réels

Comme pour **INTEGER**, **REAL** dépend de l'ordinateur utilisé. Pour plus de sûreté, on peut distinguer :

- **SINGLE** correspondant aux classes de réels de valeur absolue comprise entre $1,5 \cdot 10^{-45}$ et $3,4 \cdot 10^{38}$ avec 7 ou 8 chiffres significatifs ;
- **DOUBLE** correspondant aux classes de réels de valeur absolue comprise entre $5,0 \cdot 10^{-324}$ et $1,7 \cdot 10^{308}$ avec 15 ou 16 chiffres significatifs ;

1 1 3 CHAR

Chaque caractère est associé à un codage binaire, en général sur 8 bits (un octet) ce qui donne $2^8 = 256$ caractères possibles.
En fait, un américain a mis au point en 1961 l'ASCII (on prononce « aski ») pour « American Standard Code for Information Interchange » qui codait 128 caractères sur 7 bits ce qui est suffisant pour les américains qui n'ont pas de lettres accentuées. Il a ensuite fallu étendre ce code pour les langues comportant des caractères spéciaux et là, les normes diffèrent, ce qui crée des problèmes de compatibilité.
Depuis quelques années, le codage *Unicode* tente de remédier à ce problème en codant tous les caractères existant. C'est le standard utilisé sur les systèmes Linux par exemple.
La plupart des langages de programmation ont une commande qui associe le code ASCII (éventuellement étendu) à un caractère. On pourra donc l'utiliser dans les algorithmes généraux.
Les opérateurs associés permettent les conversion ASCII. Il s'agit de **CHR** et de **ORD**.

```
PROGRAM conv_ascii;

VAR c : CHAR;
i,min,max : INTEGER;

BEGIN
  WRITE('code inferieur :');
  READLN(min);
  WRITE('code superieur :');
  READLN(max);
```

Caractère		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2
Code	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50
Caractère	3	4	5	6	7	8	9	:	;	<	=	>	?	@	A	B	C	D	E
Code	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69
Caractère	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Code	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88
Caractère	Y	Z	[	\	]	^	_	`	a	b	c	d	e	f	g	h	i	j	k
Code	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107
Caractère	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~
Code	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126

TABLE 1.1 – Table des caractères ASCII affichables

```
  FOR i:=min TO max DO
      WRITELN(CHR(i), ' -> ',ORD(CHR(i)));
  READLN;
END.
```

1.1.4 BOOLEAN

Rien de spécial : tout fonctionne comme sur la plupart des langages.

```
PROGRAM bissextile;

VAR m : INTEGER;
    bis : BOOLEAN;
BEGIN
 WRITE('Entrez un millesime :');
 READLN(m);
 bis:= (m>1583) AND ((((m MOD 4)=0) AND ((m MOD 100)<>0)) OR ((m
     MOD 400)=0));
 WRITELN(bis);
 READLN;
END.
```

1.1.5 Type ordinal

Les types précédemment étudiés sont ordinaux sauf bien sûr le type **REAL**. On va donc pouvoir utiliser les opérateurs **SUCC**, **PRED** et **ORD** :

```
PROGRAM ordinal;
VAR c :  CHAR;
```

Guillaume Connan, Licence Creative Commons, 2010-2011

```
BEGIN
  WRITE ('Entrez un caractere : ');
  READLN (c);
  WRITELN ('Son code ASCII est : ', ORD(c));
  WRITELN ('Son predecesseur est : ', PRED(c));
  WRITELN ('Son successeur est : ', SUCC(c));
  READLN;
END.
```

1 1 6 STRING

En français on parle de *chaîne de caractère*. N'importe quoi comportant au plus 255 caractères écrits entre deux apostrophes est une chaîne de caractère.
Ce type n'est pas ordinal mais on peut utiliser les opérateurs de comparaison. C'est l'ordre lexicographique basé sur les codes ASCII qui est alors utilisé.
On concatène les chaînes avec **+**.
On obtient le nombre de caractères d'une chaîne avec `length`.
Pour extraire une sous-chaîne, on utilise `copy(mot,début,longueur)` qui extrait de la chaîne `mot` une sous-chaîne de longueur `longueur` à partir du caractère numéro `début`.

```
PROGRAM  exemple_copy ;

VAR mot : STRING;
      i : INTEGER;

BEGIN
  WRITE('Entrez un mot : ');
  READLN(mot);
  FOR i := 1 TO LENGTH(mot) DO
      WRITELN(COPY(mot,1,i));
  READLN
END.
```

On peut transformer un nombre en chaîne avec `STR(nombre,chaîne)`.
Par exemple `STR(112,ch)` place dans la chaîne `ch` la chaîne `112`.
Comme pour `WRITE`, on peut régler le format de la chaîne image.
Par exemple, `STR(1.23e3:6:1,ch)` placera dans `ch` la chaîne ` 1230.0`.
L'opération inverse de `STR` est `VAL(chaîne,nombre,indicateur)` où `indicateur` vaut 0 si la conversion s'est bien passée et sinon indique la position du premier caractère de la chaîne qui a posé problème.
Pour accéder au i-ème caractère d'une chaîne `ch`, on entre classiquement `ch[i]`.
Pour localiser la position d'une sous-chaîne dans une chaîne, on utilise :
`POS(sous-chaîne,chaîne)`. Le nombre renvoyé sera le rang où commence la sous-chaîne. Si la chaîne ne contient pas la sous-chaîne, le nombre renvoyé sera 0.
Pour supprimer une sous-chaîne d'une chaîne, on utilise :
`DELETE(chaîne,début,longueur)` qui efface une sous-chaîne de longueur `longueur` à partir du caractère numéro `début`.

Guillaume Connan, Licence Creative Commons, 2010-2011

1 1 7 ARRAY

C'est un tableau d'une ou plusieurs lignes de longueur fixe constituées d'éléments du même type.
Par exemple :

```
PROGRAM ex_array;

TYPE Vecteur = ARRAY[1..3] OF REAL;
VAR V : Vecteur;
    i : BYTE;

BEGIN
  WRITELN('Entrez les coordonnees du vecteur : ');
  FOR  i:=1 TO 3 DO READLN(V[i]);
  WRITELN('Son abscisse est : ',V[1]:2:1);
  WRITELN('Son ordonnee est : ',V[2]:2:1);
  WRITELN('Sa cote est : ',V[3]:2:1);
  WRITELN('Sa norme est environ : ',sqrt(sqr(V[1])+ sqr(V[2])+sqr
      (V[3])):2:2);
  READLN
END.
```

Voici maintenant un exemple de tableau contenant plusieurs lignes :

```
PROGRAM ex_array2;

TYPE Matrice = ARRAY[1..2,1..2] OF SINGLE;
VAR M   : Matrice;
    i,j : BYTE;
    determinant : SINGLE;

BEGIN
  WRITELN('Entrez les coordonnees des deux vecteurs : ');
  FOR  i:=1 TO 2 DO
    FOR j:=1 TO 2 DO READLN(M[i,j]);

  determinant:=M[1,1]*M[2,2]-M[1,2]*M[2,1];
  IF determinant=0 THEN
     WRITELN('Les vecteurs sont colineaires.')
  ELSE
     WRITELN('Les vecteurs ne sont pas colineaires.');
     WRITELN('Leur determinant vaut : ',determinant:2:1);
  READLN
END.
```

1 1 8 RECORD

Ce type permet de créer des sortes de tableaux dont les lignes peuvent être de longueurs et de types différents :

```
PROGRAM ex_record;

TYPE Jour = WORD;
     Mois = (Janvier,Fevrier,Mars,Avril,Mai,Juin,Juillet,Aout,
         Septembre,Octobre,Novembre,Decembre);
     Date = RECORD
       day : 1..31;
       month : Janvier..Decembre;
       year : WORD;
     END;

VAR  d : Date;

BEGIN
  WRITELN('Entrez le jour : ');
  READLN(d.day);
  WRITELN('Entrez le mois : ');
  READLN(d.month);
  WRITELN('Entrez l annee : ');
  READLN(d.year);
  WRITELN('La date est le : ',d.day,'/',ORD(d.month)+1,'/',d.year
      );
  READLN
END.
```

1 1 9 SET

Il s'agit de collections non ordonnée : des ensembles au sens mathématique du terme.
On peut les déclarer à l'aide de **SET** ou d'une collection entre crochets.

```
CONST chiffres : SET OF CHAR = [0..9];
CONST voyelles = ['a', 'e', 'i', 'o', 'u', 'y'];
TYPE lettres = SET OF CHAR;
```

On peut réunir des ensembles avec **+**, obtenir leur intersection avec *****, leur différence avec **-**. On peut également les comparer à l'aide de **=** pour l'égalité et de **<=** pour l'inclusion ou **>=**.

1 2 Priorité des opérateurs

Dans le doute, on met autant de paires de parenthèses que l'on souhaite mais il faut savoir que PASCAL suit l'ordre suivant :

1. L'opérateur unaire **-** qui représente le passage à l'opposé ;
2. NOT ;
3. * / DIV MOD AND ;
4. + - OR XOR ;
5. = < <= > >= <> IN

Guillaume Connan, Licence Creative Commons BY: $ C , 2010-2011

1 3 Structures conditionnelles

Vous connaissez le fameux **IF...THEN...ELSE**.
Il existe aussi l'instruction **CASE** :

```
PROGRAM example_case;

VAR n : INTEGER;

BEGIN
 WRITE(' entrez un nombre entier : ');
 READLN(n);
  CASE n OF
     0,1,2 : WRITELN('peut mieux faire');
     3..10 : WRITELN('insuffisant');
     11..17: WRITELN('pas mal');
  ELSE
     WRITELN('correct');
  END;
 READLN;
END.
```

1 4 Structures de répétition

1 4 1 FOR

La structure est globalement **FOR...TO...DO...**.
Cependant, il y a quelques petites particularités concernant l'incrémentation.
Voici trois exemples parlant d'eux-mêmes :

```
PROGRAM pour1;

VAR c : CHAR;

BEGIN
  FOR c:= 'a' TO 'z' DO
      WRITE(c);
  READLN
END.
```

Puis :

```
PROGRAM pour2;

VAR c : CHAR;

BEGIN
  FOR c:= 'Z' DOWNTO 'A' DO
      WRITE(c);
  READLN
END.
```

Guillaume Connan, Licence Creative Commons BY: $ ©, 2010-2011

Et enfin :

```
PROGRAM pour3;

VAR i,n : INTEGER;

BEGIN
  i:=66;
  FOR n:= 1 TO 13 DO
      BEGIN
       WRITE(CHR(i),'-');
       i:=i+2;
      END;
  WRITE('fin');
  READLN
END.
```

1 4 2 WHILE

La syntaxe est **WHILE...DO...** sans surprise.

```
PROGRAM tantque;
VAR i : INTEGER;
BEGIN
  i:=66;
  WHILE i<=90 DO
      BEGIN
       WRITE(CHR(i),'-');
       i:=i+2;
      END;
  WRITELN('fin');
  WRITELN('i vaut maintenant ', i);
  READLN
END.
```

1 4 3 REPEAT

Une petite nouveauté dont la syntaxe est **REPEAT...UNTIL...** et qui ressemble à **WHILE** avec cependant une petite différence :

```
PROGRAM repete1;
VAR i : INTEGER;
BEGIN
  i:=66;
  REPEAT
       BEGIN
       WRITE(CHR(i),'-');
       i:=i+2 ;
       END;
  UNTIL i>90;
```

```
  WRITELN('fin');
  WRITELN('i vaut maintenant', i);
  READLN
END.
```

On l'utilise souvent lorsqu'on ignore a priori le nombre de répétitions :

```
PROGRAM repete2;
VAR i : INTEGER;
BEGIN
  REPEAT
        BEGIN
        WRITELN('Entrez un nombre positif : ');
        READLN(i);
        WRITELN('Merci, c est gentil');
        END
  UNTIL i>10;
  WRITELN('C est fini maintenant');
  READLN
END.
```

1 5 Types définis par l'utilisateur

1 5 1 Types énumérés

```
PROGRAM type_ord;

TYPE couleur=(trefle,carreau,coeur,pique);
VAR carte : couleur;

BEGIN
  FOR carte := trefle TO pique DO
    BEGIN
      IF (carte=trefle) OR (carte=pique)
      THEN WRITELN('Une carte de ',carte,' est noire.')
      ELSE WRITELN('Une carte de ',carte,' est rouge.')
    END;
  READLN
END.
```

```
PROGRAM type_ord2;

TYPE jour=(lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
VAR date : jour;

BEGIN
  FOR date := lundi TO samedi DO
    BEGIN
```

Guillaume Connan, Licence Creative Commons, 2010-2011

```
      WRITE('Le ',date,' est le ',ORD(date)+1, 'e jour de la
          semaine.');
      WRITELN(' Il est suivi par ',SUCC(date),'.');
    END;
  READLN
END.
```

```
PROGRAM type_ord3;

TYPE jour=(lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche);
      boulot=lundi..vendredi;
VAR date : boulot;

BEGIN
  FOR date := boulot(0) TO boulot(4) DO
    BEGIN
      WRITELN('Le ',date,' est le ',ORD(date)+1, 'e jour de
          boulot de la semaine.');
    END;
  READLN
END.
```

1 6 Récursion

Définition : si vous ne comprenez toujours pas le sens du mot « récursion », relisez cette phrase.

Voici un exemple qui illustre l'enchaînement des calculs :

```
PROGRAM facto1;

VAR k : WORD;

FUNCTION fac ( n: INT64) : INT64;
BEGIN
  WRITELN('*** entree dans fac ; n= ',n);
  IF n=1 THEN fac:=1
  ELSE fac:=n*fac(n-1);
  WRITELN('--- sortie de fac ; n= ', n, ' et fac = ',fac);
END;

BEGIN
  WRITELN('Entrez l entier : ');
  READLN(k);
  WRITELN('Voici les etapes de calcul de ',k,'!');
  WRITELN(fac(k) );
  READLN
END.
```

Guillaume Connan, Licence Creative Commons, 2010-2011

2 Correction et terminaison d'un algorithme

Cela correspond à la *démonstration* en mathématique. Nous ne ferons que survoler ces notions de *génie logiciel* dans des cas simples.
Nous abuserons du résultat suivant :

Théorème 1 - 1

Suite strictement décroissante d'entiers naturels
Toute suite strictement décroissante d'entiers naturels converge vers 0.

Toutes nos preuves reprendront ce principe qui peut en fait être généralisé à un ensemble muni d'une relation d'ordre : sous certaines conditions (toute partie non vide admet un élément minimal) on parle d'*ordre bien fondé*.

2 1 Algorithmes impératifs

La notion centrale est l'**invariance de boucle**. Une étude plus approfondie nécessite l'étude de la logique des prédicats qui sera menée en deuxième année.
Un *invariant de boucle* est une propriété qui est vraie :

- avant d'entrer dans la boucle (avant la première itération) ;
- à chaque nouvelle itération ;
- en sortie de boucle.

Par exemple, considérons un programme de calcul de somme :

```
PROGRAM calcul_somme;

VAR Nmax,n,S : QWORD;

BEGIN
WRITE('Entrez l entier superieur :');
READLN(Nmax);
 n:=0;
 S:=0;
 WHILE n<=Nmax DO
   BEGIN
    S:=S+n;
    n:=n+1
   END;
WRITE('La somme des ',Nmax,' premiers entiers naturels vaut ',S);
READLN
END.
```

Soit la propriété $\mathfrak{p}(n)$: « après n itérations de la boucle, la valeur de S est $\sum_{k=0}^{n} k$ et $n \leqslant \mathrm{N}$ ».
La preuve par récurrence est assez immédiate. L'algorithme est donc *correct*.
Occupons-nous maintenant de sa *terminaison*. En général, ce problème est indécidable (on ne peut pas répondre par « oui » ou « non »). Mais dans certains cas, on peut définir une condition suffisante de terminaison : il s'agit de déterminer une

Guillaume Connan, Licence Creative Commons BY: $\circledS$ $\circledcirc$, 2010-2011

fonction à valeurs entières, dont les variables sont celles de l'algorithme, qui soit strictement monotone, bornée et dont les valeurs sont atteintes à chaque itération.

En général, c'est la fonction qui définit le nombre d'itérations restant à effectuer.

Par exemple, dans le cas précédent, ce pourrait être la fonction F :

$$F: (S, N, n) \longmapsto N - n$$

qui est strictement décroissante, à valeurs entières et bornée par 0.

2 2 Algorithmes récursifs

2 2 1 Généralités

Le principe est un peu le même mais on parle cette fois l'**invariant d'appel récursif**.

Reprenons le cas de la somme :

```
PROGRAM calcul_somme_rec;

VAR Nmax: QWORD;

FUNCTION som_ent(n:QWORD):QWORD;
  BEGIN
    IF n=0 THEN som_ent:=0
    ELSE som_ent:=som_ent(n-1)+n
  END;

BEGIN
WRITE('Entrez l entier superieur : ');
READLN(Nmax);
WRITE('La somme des ',Nmax,' premiers entiers naturels vaut ',
    som_ent(Nmax));
READLN
END.
```

La propriété $\mathfrak{P}(n)$ est : « `som_ent(n)` $= \sum_{k=0}^{n} k$ ».

La preuve par récurrence est assez immédiate.

Pour ce qui est de la terminaison, le principe est toujours le même ; cependant, dans la plupart des cas, la fonction à exhiber sera plus immédiate et ne dépendra que de n. Ici, c'est clairement la fonction $\mathfrak{Id}_{\mathbb{N}}: n \mapsto n$ qui est strictement décroissante et minorée par 0.

Danger

Attention ! Ne pas trouver de fonction F convenant ne prouve pas que l'algorithme ne termine pas. C'est le cas par exemple de l'algorithme de Syracuse au sujet duquel on ne sait toujours pas conclure :

```
Fonction syracuse(n : entier naturel) : entier naturel
Début
  Si n<>1 Alors
    Si n est pair Alors
      Retourner syracuse(n/2)
    Sinon
      Retourner syracuse(3*n+1)
    FinSi
  Sinon
    Retourner 1
  FinSi
Fin
```

2 2 2 Algorithme récursif terminal

On appelle ainsi les algorithmes dont l'appel récursif ne figure pas comme argument ni de l'appel lui-même ni d'une autre fonction.
Dans le cas précédent du calcul de somme, l'appel récursif de **som_ent** était *enrobé* dans la fonction $F\colon (X,Y)\mapsto X+Y$.
On avait en effet **som_ent(n):=F(som_ent(n-1),n)**.
On peut transformer un peu l'algorithme pour le rendre terminal :

```
PROGRAM calcul_somme_term;

VAR Nmax: QWORD;

FUNCTION somme(n:QWORD):QWORD;
 FUNCTION som_ent(x,y:QWORD):QWORD;
   BEGIN
    IF x=n THEN som_ent:=y
    ELSE som_ent:=som_ent(x+1,y+x+1)
   END;
  BEGIN
   somme:=som_ent(0,0)
  END;

BEGIN
 WRITE('Entrez 1 entier superieur : ');
 READLN(Nmax);
 WRITE('La somme des ',Nmax,' premiers entiers naturels vaut ',
     somme(Nmax));
 READLN
END.
```

Cependant, sur Pascal, cela ne change rien. Ce n'est efficace que dans le cas d'un langage moderne et fonctionnel comme par exemple CAML...
Occupons-nous cependant de la preuve de la correction et de la terminaison de cet algorithme qui est moins triviale que la précédente.
L'appel de **somme(n)** déclenche l'appel de **som_ent(0,0)**. Notons $(x_i, y_i)_{i\in\mathbb{N}}$ la suite des couples d'arguments passés à **som_ent** suite à l'appel **som_ent(0,0)**.
On a $(x_0, y_0) = (0,0)$.

On peut démontrer que $(x_i, y_i) = (i, S_i)$ pour tout entier naturel i, avec $S_i = \sum_{k=0}^{i} k$ par récurrence (faites-le !).
Pour prouver la terminaison, on montre que la fonction $F : i \mapsto n - i$ est strictement décroissante tout au long des appels récursifs, à valeurs entières et minorée par 0.

3 Récursion mutuelle

On peut définir la parité d'un entier ainsi :

- 0 est pair ;
- n est pair signifie que $n-1$ est impair ;
- 1 est impair ;
- n est impair signifie que $n-1$ est pair.

On parle de récursion mutuelle car la parité et l'imparité sont définie récursivement l'une en fonction de l'autre.
Sur PASCAL, le mot magique est **FORWARD** :

```
PROGRAM mutuelle_rec;

VAR n:WORD;

FUNCTION impair(n:WORD):BOOLEAN; FORWARD;

FUNCTION pair(n:WORD):BOOLEAN;
BEGIN
  IF n=0
  THEN pair:=TRUE
  ELSE pair:=impair(n-1)
END;

FUNCTION impair(n:WORD):BOOLEAN;
BEGIN
  IF n=0
  THEN impair:=FALSE
  ELSE impair:=pair(n-1)
END;

BEGIN
```

```
  WRITE('Entrez un entier naturel : ');
  READLN(n);
  IF pair(n)
  THEN WRITELN(n, ' est pair.')
  ELSE WRITELN(n, ' est impair.');
  READLN
END.
```

4 Diviser pour régner

EXERCICES

1 - 1 L'addition n'est plus associative ? !

Pour tout $p \in \mathbb{N}^*$, on pose $S_p = \sum_{n=1}^{p} \frac{1}{n^3}$.

Écrire deux fonctions calculant une valeur approchée de S_p, l'une en accumulant les termes suivant les valeurs croissantes de n, l'autre suivant les valeurs décroissantes de n. Comparez les résultats obtenus pour diverses valeurs de p en utilisant pour les valeurs des sommes le type `SINGLE` puis le type `DOUBLE`.

Sur un logiciel de calcul formel on obtient :

$$S_{10^{15}} \approx 1,202056903159594285399738161 51$$

1 - 2 Suite de Syracuse

On attend toujours la preuve de la convergence de la suite $(a_n)_{n\in\mathbb{N}}$ définie par :

$$a_0 = a \quad \text{et} \quad \forall n \in \mathbb{N} \quad a_{n+1} = \begin{cases} \frac{a_n}{2} & \text{si } n \text{ est pair} \\ 3a_n + 1 & \text{sinon} \end{cases}$$

Déterminez des programmes recevant comme paramètre la valeur a et affichant la séquence des a_n jusqu'au premier égal à 1.

Déterminez un programme renvoyant le plus petit n tel que $a_n = 1$ pour un a donné.

Déterminez la ou les valeur(s) de a sur $[\![1;10\,000]\!]$ telle(s) que le plus petit n tel que $a_n = 1$ soit maximal.

1 - 3 Triangle de Pascal

Affichez un triangle de PASCAL en n'utilisant comme opération arithmétique que des additions.

On obtiendra par exemple :

```
Indiquez le nombre de lignes  : 6

  1
  1   1
  1   2   1
  1   3   3   1
  1   4   6   4   1
  1   5  10  10   5   1
```

Modifiez légèrement le programme précédent en remplaçant les nombres impairs par une étoile `*` et les nombres pairs par une espace. Qu'observez-vous pour 64 lignes par exemple ?

1 - 4 Crible d'Ératosthène

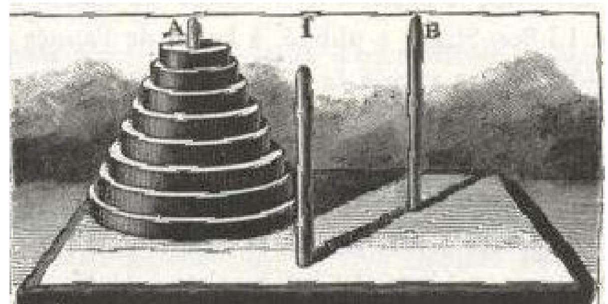
C'est comme au collège : vous prenez les premiers nombres entiers naturels et vous rayez les multiples de 2, de 3, etc. jusqu'à ne conserver que les nombres premiers.

Inspirez-vous de cette idée pour obtenir une liste des entiers premiers entre 2 et un entier n donné ainsi que le nombres d'entiers premiers inférieurs à n (on note souvent ce nombre $\pi(n)$).

1 - 5 Les tours de Hanoi

Nous en avons déjà parlé. Vous pouvez observer la résolution du problème à quatre disques à cette adresse http ://www.youtube.com/watch ?v=aGlt2G-DC8c

La situation de départ est la suivante :



Vous pouvez montrer que le nombre minimum de coups pour bouger n disques suit la relation de récurrence :

$$C_0 = 0, \ \forall n \in \mathbb{N}^*, \ C_n = 2C_{n-1} + 1$$

Déterminez un programme utilisant une fonction récursive pour détailler chaque coup permettant

de résoudre le problème à n disques en indiquant également le nombre total de coups.

1 - 6 Conversions d'entiers

Déterminez un programme convertissant un entier dans une base quelconque (entre 2 et 10). Vous en donnerez une version impérative et une version récursive.

1 - 7 Analyse de fonctions récursives

Commentez les fonctions suivantes :

```
FUNCTION facto(n:WORD):WORD;
BEGIN
 IF n=0
 THEN facto:=1
 ELSE facto:=facto(n+1)/(n+1)
END;
```

```
FUNCTION Gibbs(a,b:WORD):WORD;
BEGIN
  IF a=0
  THEN Gibbs:=1
  ELSE Gibbs:=Gibbs(a-1,Gibbs(a,b))
END;
```

1 - 8 Récursion multiple (e3a 2009)

On se donne deux entiers $k \in \mathbb{Z}$ et $n \in \mathbb{N}$. Le coefficient binomial associé à ces deux entiers est :

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!} & \text{si } 0 \leqslant k \leqslant n \\ 0 & \text{sinon} \end{cases}$$

On a alors la formule de PASCAL :

$$\forall n \geqslant 1,\ \forall k \in \mathbb{Z},\ \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

1. *Dans cette question, on n'utilisera ni liste, ni tableau et la seule opération arithmétique autorisée est l'addition ; on n'effectuera donc aucune multiplication.* Écrire une fonction récursive simple calculant le coefficient $\binom{n}{k}$.

2. On pose $n = 1\,000$ et on souhaite construire un tableau t de longueur $n+1$ tel que si $k \in [\![0, n]\!]$, t$[k]$ contient la valeur de $\binom{n}{k}$.

- **a.** Expliquez brièvement pourquoi il n'est pas possible d'utiliser la fonction de la question précédente.
- **b.** Écrire un programme renvoyant le tableau convenablement rempli.

1 - 9 Racine carrée entière (e3a 2009)

On s'intéresse au problème suivant : étant donné un entier $n \geqslant 1$, on souhaite déterminer un entier positif s_n de sorte que s_n soit « proche » de $\sqrt{n}$. Nous étudierons différentes approches pour résoudre ce problème. On se donne les contraintes suivantes :

1. on ne travaillera qu'avec des entiers relatifs ;

2. les opérations arithmétiques autorisées sur les entiers sont :

- **a.** l'addition de deux entiers a et b : **a+b** ;
- **b.** la soustraction de deux entiers a et b : **a-b** ;
- **c.** la multiplication de deux entiers a et b : **a*b** ;
- **d.** la division euclidienne d'un entier positif a par un entier strictement positif b : le reste de la division est obtenu par **a MOD b** et le quotient par **a DIV b**.

Dans la suite, si a et b sont deux entiers avec b_neq0, l'écriture $\frac{a}{b}$ désigne le quotient **rationnel** de a par b ; si x est un réel, $\lfloor x \rfloor$ désigne la partie entière de x et $\lceil x \rceil$ désigne le plafond de x : le plus petit entier supérieur ou égal à x.

1. Algorithme naïf

Dans cette partie, étant donné un entier $n \geqslant 1$, on cherche à déterminer le plus grand entier s_n tel que $s_n^2 \leqslant n$. Pour cela, on passe en revue les entiers 1, 2, 3,... jusqu'à trouver s_n.

- **a.** Écrire une fonction **isqrt_naif** prenant n en argument et renvoyant la valeur s_n en utilisant le principe ci-dessus.
- **b.** Déterminer la complexité de votre programme.

2. Une méthode dichotomique

Dans cette partie, on reprend le problème de la partie précédente. Afin d'obtenir une méthode plus efficace, on considère le programme suivant :

```
FUNCTION isqrt(n:LONGINT):LONGINT;
  VAR x,y,z:LONGINT;
  BEGIN
    x:=1; y:=n;
    WHILE (y-x>1) DO
      BEGIN
       z:=(x+y) DIV 2;
       IF z*z>n THEN
         y:=z
       ELSE
         x:=z
      END;
    isqrt:=x
  END;
```

Si $k \geqslant 1$, on note x_k et y_k les valeurs des variables x et y à la fin de la k-eme itération de la boucle **WHILE**. On convient que $x_0 = 1$ et $y_0 = n$.

a. Expliciter les exécutions du programme ci-dessus sur l'entier 15. (On donnera en particulier les valeurs successives des variables x, y et z.)

b. Démontrer la terminaison du programme.

c. Justifier la correction du programme.

d. i. Prouver l'inégalité suivante :

$$y_k - x_k \leqslant \frac{y_0 - x_0 - 1}{2^k} + 1$$

ii. En déduire une majoration de la complexité du programme.

e. Réécrire la fonction **isqrt** en utilisant de la récursivité.

1 - 10 Nombres d'Armstrong (e3a 2008)

Un nombre d'ARMSTRONG est un nombre qui est égal à la somme des cubes des chiffres de son écriture en base 10 ; par exemple 153 est un nombre d'ARMSTRONG puisque $153 = 1^3 + 5^3 + 3^3$. Écrire un programme qui calcule tous les nombres d'ARMSTRONG de trois ou quatre chiffres.

1 - 11 Nombres parfaits (e3a 2007)

Soit n un entier naturel. On dit que n est un nombre parfait si $2n$ est la somme des entiers naturels diviseurs de n. Par exemple, l'entier 6 est un nombre parfait puisque $2 \cdot 6 = 12 = 6+3+2+1$.
Écrire un programme qui détermine la liste des nombres parfaits n tels que $n \leqslant 9\,999$.

1 - 12 Persistance (e3a 2007)

Un entier naturel n étant donné, on calcule le produit **prod(n)** de ses chiffres dans son écriture en base 10 puis le produit des chiffres de **prod(n)** et on recommence ainsi l'application de **prod** jusqu'à obtenir un chiffre entre 0 et 9. Le nombre minimal de fois où on applique **prod** pour transformer n en un chiffre entre 0 et 9 est appelé la *persistance* de n. Par exemple, la persistance de 9 est égale à 0, celle de 97 est égale à 3 car **prod**$(97) = 9 \cdot 7 = 63$, **prod**$(63) = 6 \cdot 3 = 18$, **prod**$(18) = 1 \cdot 8 = 8$. et celle de 9 575 est égale à 5.
Écrire un programme qui calcule le plus petit entier naturel de persistance 5.

DES SOLUTIONS

1 - 1

```
PROGRAM exo1_1;

TYPE Reel = SINGLE;
     Entier=LONGWORD;
VAR k : Entier;

FUNCTION terme ( n: Entier) : Reel;
BEGIN
  terme:=1/(n*n*n)
  END;

FUNCTION som_cr(p:Entier):Reel;
 VAR s:Reel;
    n:Entier;
BEGIN
 s:=0;
 FOR n:=1 TO p DO s:=s+terme(n);
 Som_cr:=s;
END;

FUNCTION som_dec(p:Entier):Reel;
 VAR s:Reel;
    n:Entier;
BEGIN
 s:=0;
 FOR n:=p DOWNTO 1 DO s:=s+terme(n);
 Som_dec:=s;
END;

BEGIN
  WRITELN('Entrez le terme max :  ');
  READLN(k);
  WRITELN('Somme_croissante(',k,') = ',som_cr(
      k));
  WRITELN('Somme_decroissante(',k,') = ',
      som_dec(k));
  READLN
END.
```

1 - 2

```
PROGRAM exo1_2_a;

VAR a:LONGWORD;

PROCEDURE  Syracuse(a:LONGWORD);
VAR u : LONGWORD;
BEGIN
  u:=a;
  WHILE u<>1 DO
    BEGIN
      WRITE(u:5);
      IF ODD(u)
         THEN u:=3*u+1
         ELSE u:=u DIV 2;
    END;
  WRITELN(1:5)
END;

BEGIN
 WRITE('Entrez le premier terme :  ');
 READLN(a);
 Syracuse(a);
 READLN
END.
```

```
PROGRAM exo1_2_b;

VAR nmax,a,n : WORD;

FUNCTION  Premier_Syracuse(a:LONGWORD):WORD;
VAR u : LONGWORD;
    n : WORD;
BEGIN
  n:=0;
  u:=a;
  WHILE u<>1 DO
    BEGIN
      n:=n+1;
      IF ODD(u)
         THEN u:=3*u+1
         ELSE u:=u DIV 2;
    END;
  Premier_Syracuse:=n
END;

BEGIN
 nmax:=0;
 FOR a:=1 TO 10000 DO
   BEGIN
     n:=Premier_Syracuse(a);
     IF n>=nmax THEN
       BEGIN
         nmax:=n;
         WRITELN(a,'->',n)
       END;
   END;
 READLN
END.
```

1 - 3

```
PROGRAM exo1_3;

CONST Nmax=15;
```

```
TYPE Entier=LONGWORD;
VAR nl : Entier; {nb de lignes}
     i : Entier; {indice de ligne}
     j : Entier; {indice de colonne}
     T : ARRAY[0..Nmax-1,0..Nmax-1] OF Entier;
           {Le tableau contenant le triangle}


BEGIN
  WRITE('Indiquez le nombre de lignes  : ');
  READLN(nl);
  WRITELN;
  WRITELN;
  T[0,0]:=1;
  WRITELN(T[0,0]:5);
  FOR i:=1 TO nl-1 DO
    BEGIN
      T[i,i]:=1;
      T[i,0]:=1;
      WRITE(T[i,0]:5);
      FOR j:=1 TO i-1 DO
        BEGIN
          T[i,j]:=T[i-1,j]+T[i-1,j-1];
          WRITE(T[i,j]:5);
        END;
      WRITELN(T[i,i]:5);
    END;
  READLN
END.
```

```
PROGRAM exo1_3_b;

CONST Nmax=65;
TYPE Entier=LONGWORD;
VAR nl : Entier; {nb de lignes}
     i : Entier; {indice de ligne}
     j : Entier; {indice de colonne}
     T : ARRAY[0..Nmax-1,0..Nmax-1] OF Entier;
           {Le tableau contenant le triangle}


BEGIN
  WRITE('Indiquez le nombre de lignes  : ');
  READLN(nl);
  WRITELN;
  WRITELN;
  T[0,0]:=1;
  WRITELN('*');
  FOR i:=1 TO nl-1 DO
    BEGIN
      T[i,i]:=1;
      T[i,0]:=1;
      WRITE('*');
      FOR j:=1 TO i-1 DO
        BEGIN
          T[i,j]:=T[i-1,j]+T[i-1,j-1];
          IF ODD(T[i,j])
            THEN WRITE('*')
            ELSE WRITE(' ')
```

```
        END;
      WRITELN('*');
    END;
  READLN
END.
```

1 - 4

```
PROGRAM exo1_4;

CONST Nmax=1000000;
TYPE tableau = ARRAY[1..Nmax]  OF LONGWORD;
VAR N,k,naff:LONGWORD;
    raye : tableau;

PROCEDURE erato;
VAR i,k    : LONGWORD;
BEGIN
  FOR i:=1 TO Nmax DO raye[i]:=i;
  raye[1]:=0;
  i:=2;
  WHILE i*i<=Nmax DO
    BEGIN
      IF raye[i]>=1 THEN
        BEGIN
         FOR k:=2 TO Nmax DIV i DO
          raye[i*k]:=0
        END;
      i:=i+1
    END;
END;

BEGIN
  naff:=0;
  WRITE('Entrez l entier maximum :  ');
  READLN(N);
  erato;
  WRITELN;
  WRITELN('Les nombres premiers inferieurs a '
       ,N,' sont :');
  WRITELN;
  FOR k:=1 TO N DO
    BEGIN
      IF raye[k]>0 THEN
        BEGIN
         WRITE(k:5);
         naff:=naff+1;
         IF (naff MOD 15 = 0) THEN WRITELN
        END;
    END;
WRITELN;
WRITELN;
WRITELN;
WRITELN('Il y a ',naff,' nombres premiers
     inferieurs a ',N);
READLN
END.
```

1 - 5

```
PROGRAM exo1_5;

USES crt;

VAR n,i : WORD;
    A,B,C : STRING;

PROCEDURE  deplace(x,y:STRING);
BEGIN
  i:=i+1;
  WRITE('Deplacez le disque en ',x,' sur le
      piquet ',y);
  READLN
END;

PROCEDURE hanoi(n:WORD;A,B,C:STRING);
BEGIN
  IF n=1
    THEN
     deplace(A,C)
    ELSE
     BEGIN
      hanoi(n-1,A,C,B);
      deplace(A,C);
      hanoi(n-1,B,A,C);
     END
END;

BEGIN
  i:=0;
  A:='A';B:='B';C:='C';
  WRITE('Combien de disques : ');
  READLN(n);
  hanoi(n,A,B,C);
  WRITELN('Et voila ! Les disques ont ete
      deplaces en ',i,' coups.');
  READLN
END.
```

1 - 6

En impératif :

```
PROGRAM exo1_6a;

CONST NbChifMax = 63;

VAR n : QWORD; {Nombre a convertir}
    quotient : QWORD;
    reste : QWORD;
    b : BYTE; {base}
    conv : ARRAY[1..NbChifMax] OF BYTE; {
        nombre converti sous forme de liste}
    NbChif : BYTE; {Nombre effectif de
        chiffres}
    i : WORD; {indice}

BEGIN
  {--- lecture des donnees ---}
  WRITE('Entrez le nombre a convertir : ');
  READLN(n);
  WRITE('ENtrez la base voulue (entre 2 et 10)
       : ');
  READLN(b);
  {--- conversion ---}
  NbChif:=1;
  reste:= n MOD b;
  quotient:= n DIV b;
  conv[NbChif]:=reste;
  WHILE quotient>0 DO
     BEGIN
      reste:=quotient MOD b;
      quotient:=quotient DIV b;
      NbChif:=NbChif+1;
      conv[NbChif]:=reste;
     END;
  {--- resultat ---}
  WRITE(n, ' en base ',b,' : ');
  FOR i:=NbChif DOWNTO 1 DO
    WRITE(conv[i]:2);
  READLN
END.
```

Autre version en utilisant des chaînes :

```
PROGRAM exo1_6a_bis;

VAR n : QWORD; {Nombre a convertir}
    quotient : QWORD;
    reste : QWORD;
    b : BYTE; {base}
    conv : STRING; {nombre converti sous forme
         de chaine}

BEGIN
  {--- lecture des donnees ---}
  WRITE('Entrez le nombre a convertir : ');
  READLN(n);
  WRITE('ENtrez la base voulue (entre 2 et 10)
       : ');
  READLN(b);
  {--- conversion ---}
  reste:= n MOD b;
  quotient:= n DIV b;
  conv:=CHR(ORD('0')+reste);
  WHILE quotient > 0 DO
     BEGIN
      reste:=quotient MOD b;
      quotient:=quotient DIV b;
      conv:=CHR(ORD('0')+reste)+conv
     END;
  {--- resultat ---}
  WRITELN;
  WRITE(n, ' en base ',b,' : ',conv);
  READLN
END.
```

En récursif :

```
PROGRAM exo1_6b;

VAR n : QWORD; {Nombre a convertir}
    b : BYTE; {base}

FUNCTION conv(n:QWORD;b:BYTE):STRING;
 BEGIN
  IF n<2 THEN conv:=CHR(ORD('0')+n)
  {on rajoute ORD('0') pour obtenir le bon
      code ASCII}
  ELSE conv:=conv(n DIV b,b)+CHR(ORD('0')+n
      MOD b);
 END;

BEGIN
  {--- lecture des donnees ---}
  WRITE('Entrez le nombre a convertir : ');
  READLN(n);
  WRITE('Entrez la base voulue (entre 2 et 10)
      : ');
  READLN(b);
  {--- resultat ---}
  WRITE(n, ' en base ',b,' : ',conv(n,b));
  READLN
END.
```

1 - 7

Problèmes de terminaison.

1 - 8

Sans multiplication :

```
PROGRAM exo1_8b;

VAR n:LONGWORD;
    k:LONGINT;


FUNCTION pasc(n:LONGWORD;k:LONGINT):QWORD;
BEGIN
  IF (k<0) OR (k>n)
  THEN pasc:=0
  ELSE
   BEGIN
     IF (k=0) OR (k=n)
     THEN pasc:=1
     ELSE pasc:=( (n-k+1)*pasc(n,k-1) ) DIV k
   END;
END;


BEGIN
  WRITE('Entrez un entier naturel : ');
  READLN(n);
  WRITE('Entrez un entier : ');
  READLN(k);
  WRITELN(k,' parmi ',n,' vaut ',pasc(n,k));
  READLN
END.
```

Avec multiplication :

```
PROGRAM exo1_8b;

VAR n:LONGWORD;
    k:LONGINT;


FUNCTION pasc(n:LONGWORD;k:LONGINT):QWORD;
BEGIN
  IF (k<0) OR (k>n)
  THEN pasc:=0
  ELSE
   BEGIN
     IF (k=0) OR (k=n)
     THEN pasc:=1
     ELSE pasc:=( (n-k+1)*pasc(n,k-1) ) DIV k
   END;
END;


BEGIN
  WRITE('Entrez un entier naturel : ');
  READLN(n);
  WRITE('Entrez un entier : ');
  READLN(k);
  WRITELN(k,' parmi ',n,' vaut ',pasc(n,k));
  READLN
END.
```

Le tableau :

```
PROGRAM exo1_8c;

CONST n=15;

VAR k : WORD;
    tpasc:ARRAY[0..n] OF QWORD;


BEGIN
  tpasc[0]:=1;
  tpasc[n]:=1;

  FOR k:=1 TO (n DIV 2) DO
  BEGIN
   tpasc[k]:=( (n-k+1)*tpasc[k-1] ) DIV k;
   tpasc[n-k]:=tpasc[k]
  END;

END.
```

1 - 9

```
PROGRAM exo1_9a;

VAR n,s:QWORD;

FUNCTION isqrt_naif(n:QWORD):QWORD;
  VAR s:QWORD;
  BEGIN
    s:=0;
    WHILE (s*s)<=n DO
     s:=s+1;
    isqrt_naif:=s-1
  END;

BEGIN
  WRITE('Entrez un entier naturel : ');
  READLN(n);
  s:=isqrt_naif(n);
  WRITELN('Le plus grand entier cherche est :
       ',s);
  WRITE('En effet, ',s,'^2=',s*s,'<=',n);
  WRITE(' mais (',s,'+1)^2=',(s+1)*(s+1),'>',n
       );
  READLN
END.
```

Version récursive :

```
PROGRAM exo1_9c;

VAR n,s:LONGINT;

FUNCTION isqrt_temp(x,y,n:LONGINT):LONGINT;
  BEGIN
    IF (y-x<=1) THEN
       isqrt_temp:=x
    ELSE
      BEGIN
       IF ((x+y) DIV 2)*((x+y) DIV 2) > n THEN
         isqrt_temp:=isqrt_temp(x,(x+y) DIV 2,
             n)
       ELSE
         isqrt_temp:=isqrt_temp((x+y) DIV 2,y,
             n)
      END;
  END;


FUNCTION isqrt(n:LONGINT):LONGINT;
 BEGIN
 isqrt:=isqrt_temp(1,n,n);
 END;

BEGIN
  WRITE('Entrez un entier naturel : ');
  READLN(n);
  s:=isqrt(n);
  WRITELN('Le plus grand entier cherche est :
       ',s);
  WRITE('En effet, ',s,'^2=',s*s,'<=',n);
  WRITE(' mais (',s,'+1)^2=',(s+1)*(s+1),'>',n
       );
  READLN
END.
```

1 - 10

```
PROGRAM exo1_10;

VAR l,sc,k,i:WORD;
    sn:STRING;

FUNCTION cube(n:WORD):WORD;
 BEGIN
  cube:=n*n*n;
 END;

FUNCTION test(n:WORD):BOOLEAN;
   BEGIN
     STR(n,sn);
     l:=LENGTH(sn);
     sc:=0;
     FOR k:=1 TO l DO
       sc:=sc+cube(ORD(sn[k])-ORD('0'));
     IF sc=n THEN test:=TRUE
     ELSE test:=FALSE
   END;

BEGIN
  FOR i:=100 TO 9999 DO
    IF test(i) THEN WRITELN(i);
  READLN
END.
```

1 - 11

```
PROGRAM exo1_11;

VAR n,s,k,fin,i:QWORD;

FUNCTION test(n:WORD):BOOLEAN;
BEGIN
  s:=1;
  fin:=n;
  FOR k:=2 TO fin-1 DO
    BEGIN
      IF n MOD k=0 THEN
        BEGIN
         fin:=n MOD k;
         IF k=fin THEN s:=s+k
         ELSE s:=s+k+n MOD k
        END;
    END;
  IF s=n THEN test:=TRUE
  ELSE test:=FALSE
END;
```

Guillaume Connan, Licence Creative Commons, 2010-2011

```
BEGIN
FOR i:=1 TO 9999 DO
  BEGIN
    IF test(i) THEN WRITELN(i)
  END;
READLN
END.
```

1 - 12

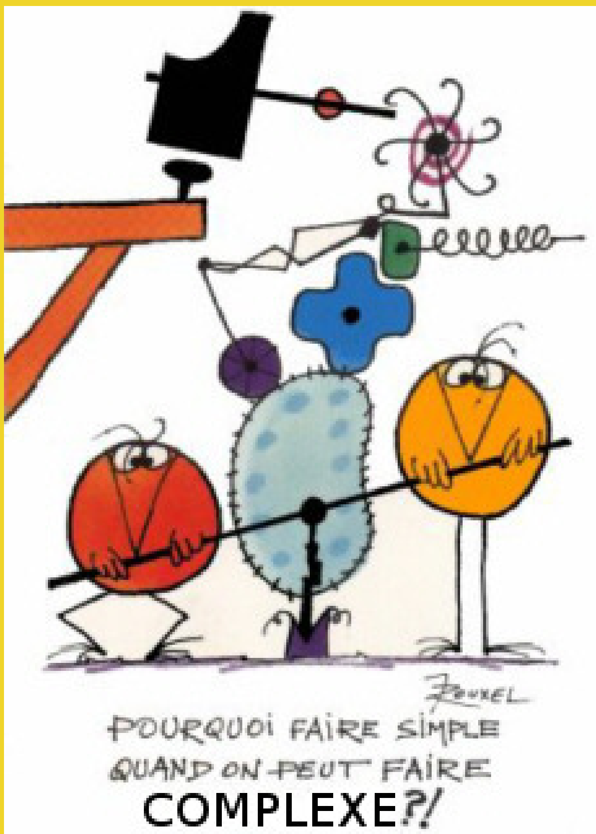
```
PROGRAM exo1_12;

VAR l,pc,k,i:WORD;
    sn:STRING;


FUNCTION prod(n:WORD):WORD;
   BEGIN
     STR(n,sn);
     l:=LENGTH(sn);
     pc:=1;
     FOR k:=1 TO l DO
       pc:=pc*(ORD(sn[k])-ORD('0'));
   prod:=pc
   END;


FUNCTION persi(n:WORD):WORD;
BEGIN
  IF prod(n)<10 THEN persi:=1
  ELSE persi:=1+persi(prod(n))
END;


BEGIN
  i:=1;
  WHILE persi(i)<5 DO i:=i+1;
  WRITELN(i);
  READLN
END.
```

Guillaume Connan, Licence Creative Commons BY: $ = , 2010-2011

OPTION

2

Complexité



1 Premières définitions

Nous étudierons la complexité uniquement en tant que succession d'opérations élémentaires, ou, plus formellement :

Définition 2 - 1

Soit un algorithme A et un argument a.
La complexité de A pour a est le nombre de fois que l'opération élémentaire choisie au préalable est exécutée lors de l'exécution de A pour l'argument a.
On la notera $K_A(a)$.

Par exemple, pour le calcul d'une factorielle :

```
Fonction facto(n : entier naturel) : entier naturel
Début
  | Si n=0 Alors
  |   | Retourner 1
  | Sinon
  |   | Retourner n*facto(n-1)
  | FinSi
Fin
```

Clairement, pour $n \geqslant 1$, $K_{\texttt{facto}}(n) = 1 + K_{\texttt{facto}}(n-1)$.
On a donc une progression arithmétique. Or $K_{\texttt{facto}}(0) = 0$ donc, pour tout entier naturel n, $K_{\texttt{facto}}(n) = n$.

Définition 2 - 2

Deux fonctions f et g de $\mathbb{N}^*$ dans $\mathbb{R}$ ont même ordre de grandeur asymptotique si, et seulement si,

$$\exists k \in \mathbb{R}_+^*, \exists k' \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^*, \forall n \in \mathbb{N}^*, n > n_0 \Longrightarrow k \cdot g(n) \leqslant f(n) \leqslant k' \cdot g(n)$$

On notera $f = \Theta(g)$ qui se lit « *f est en grand théta de g* ».

Ici, la complexité de **facto** est en $\Theta(n)$...
On classe souvent les algorithmes selon leur complexité :

- logarithmique en $\Theta(\log_2(n))$;
- linéaire en $\Theta(n)$;
- quasi-linéaire en $\Theta(n \log_2(n))$;
- quadratique en $\Theta(n^2)$;
- polynomiale en $\Theta(n^k)$ avec $k \in \mathbb{N}^*$;
- exponentielle en $\Theta(\lambda^n)$ avec $\lambda > 1$.

Il va être primordial d'évaluer la complexité d'un algorithme, même si c'est un problème souvent compliqué auquel on ne peut pas répondre précisément et qui dépend de la machine utilisée (c'est pourquoi on utilise des ordres de grandeur).

Guillaume Connan, Licence Creative Commons BY: $\circledS$ $\circledC$, 2010-2011

Par exemple, si on veut calculer le déterminant d'une matrice de taille 30 à l'aide de la définition :

$$\det A = \sum_{\sigma \in S_{30}} \varepsilon(\sigma) \cdot a_{1\sigma(1)} \cdot a_{2\sigma(2)} \cdots a_{30\sigma(30)}$$

Il s'agit de la somme de $30! \approx 2,65 \cdot 10^{32}$ termes de 25 facteurs. En supposant que le processeur soit capable de calculer 10 milliards de ces termes par secondes, cela nous prendra quand même environ huit mille milliards de siècles...
On ne s'intéressera qu'aux opérations arithmétiques basiques et on négligera les opérations d'écriture ou de lecture en mémoire (ce qui se vérifie dans la pratique).
De plus, il faudra bien parler d'opérations arithmétiques basiques (par exemple l'addition de deux nombres de 10 chiffres ou de deux nombres de 100 chiffres n'est pas effectuée de la même façon !).
On vérifie aussi expérimentalement que les temps de calcul de deux machines sont grosso modo proportionnels, ce coefficient étant « absorbé » par les coefficients du Θ. Nos calculs seront donc ainsi indépendants de la machine utilisée.
Enfin, il s'avèrera parfois difficile de calculer exactement cette complexité dans les méandres des différents tests. On distinguera alors une *complexité dans le pire des cas* et une *complexité moyenne* ou *statistique*.
On s'intéressera le plus souvent à la complexité dans le pire des cas car elle est la plus simple à calculer même si elle est le plus souvent éloignée de la réalité.

2 Premier exemple : exponentiation

2.1 Méthode naïve

```
Fonction puisr1(a,n: entiers naturels): entier naturel
Début
  Si n=0 Alors
    Retourner 1
  Sinon
    Retourner a*puis1(a,n-1)
  FinSi
Fin
```

Ici encore, la complexité est clairement en $\Theta(n)$, même avec une méthode itérative :

```
Fonction puisi1(a,n: entiers naturels): entier naturel
Variable
i,r:
Début
  r ← 1
  Pour i variantDe 1 à n Faire
    r ← a*r
  FinPour
  Retourner r
Fin
```

2 2 Diviser pour régner

On va distinguer deux cas, selon la parité de l'exposant, ce qui va nous permettre de gagner en efficacité. En effet :

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ a\cdot\left(a^{\frac{n-1}{2}}\right)^2 & \text{sinon} \end{cases}$$

```
Fonction puis2( a,n : entiers naturels ) : entier naturel
Début
    Si n=0 Alors
        Retourner 1
    Sinon
        Si n=1 Alors
            Retourner a
        Sinon
            Si n est pair Alors
                Retourner puis2(a*a,n/2)
            Sinon
                Retourner a*puis2(a*a,(n-1)/2)
            FinSi
        FinSi
    FinSi
Fin
```

Par exemple, $a^{11} = a\cdot\left(a^2\right)^5 = a\cdot a^2\cdot\left(\left(a^2\right)^2\right)^2$ ce qui entraîne 5 multiplications au lieu de 11.
De même, vérifiez que x^{27} se calcule en ... multiplications et $x^{1\,000\,000\,000}$ en 41 multiplications.
Généralisons : soit K la complexité de cet algorithme. On a $\mathrm{K}(1) = 0$.
Sinon, le nombre de multiplications effectuées pour calculer **`puis2(a,n)`** est égal au nombre de multiplications de **`puis(a,[n/2])`** auquel il faut ajouter 1 (pour **`a*a`**) si n est pair et 2 (**`a*a`** puis **`a*puis2`**) si n est impair.
Ainsi, $\mathrm{K}(1) = 0$ et $1 + \mathrm{K}([n/2]) \leqslant \mathrm{K}(n) \leqslant 2 + \mathrm{K}([n/2])$.

On montre par récurrence que la suite $(u_n)_{n\in\mathbb{N}^*}$ définie par $\begin{cases} u_1 = 0 \\ u_n = 1 + u_{[n/2]} \end{cases}$

a pour valeur $u_n = \left[\log_2(n)\right] + 1$.

De même, la suite $(v_n)_{n\in\mathbb{N}^*}$ définie par $\begin{cases} v_1 = 0 \\ v_n = 2 + v_{[n/2]} \end{cases}$

a pour valeur $v_n = 2\cdot\left[\log_2(n)\right]$.
On en déduit que

$$\forall n \in \mathbb{N},\ \left[\log_2(n)\right] \leqslant \mathrm{K}(n) \leqslant 2\left[\log_2(n)\right]$$

Ainsi, la complexité de cet algorithme est en $\Theta\left(\log_2(n)\right)$ ce qui est bien plus efficace que l'algorithme naïf : on vérifie qu'il faut entre 29 et 58 opérations pour calculer un nombre à la puissance 1 000 000 000.
Ce n'est cependant pas toujours l'algorithme le plus efficace. Par exemple, il est plus rapide de calculer a^{27} en faisant 6 multiplications au lieu de 7 :

Guillaume Connan, Licence Creative Commons, 2010-2011

$$a^{27} = \left(\left(a^3\right)^3\right)^3$$

Cependant, un algorithme qui tenterait de déterminer l'algorithme de calcul le plus efficace risque d'être beaucoup plus coûteux...

3 Recherche dans un tableau

3 1 Préliminaires

On veut rechercher si une valeur se trouve dans un tableau donné et à quel endroit. On définit un type d'éléments qui constitueront le tableau, un type d'indices et un type de tableau de taille fixée à l'avance. Ces types seront utilisés tout au long de la section.

```
CONST
  ind_max = 100;

TYPE
  element = WORD;
  indice = 1..ind_max;
  tableau = ARRAY[indice] of element;
```

3 2 Recherche séquentielle

On parcourt les éléments du tableau un par un à partir du premier.

```
PROCEDURE recherche_seq(tab : tableau;
                        debut,fin : indice;
                        valeur_cherchee : element);
VAR ici : indice;
    trouve : boolean;

{-- Si on trouve un element egal a valeur_cherchee alors ici sera
      egal a
  l'indice de cet element et trouve vaudra TRUE.

Parametres : tab (E) -> tableau dans lequel on effectue la
    recherche
             debut (E) -> indice de debut de recherche
             fin (E) -> indice de fin de recherche
             valeur_cherchee (E) -> comme son nom l'indique
             ici (S) -> indice de l'element trouve
             trouve (S) -> vrai ou faux selon que la valeur a ete
                 trouvee ou non

Exemple d'appel : recherche_seq(tab,1,10,32) --}
```

```
BEGIN
 trouve:=FALSE;
 ici:=debut;
 WHILE ((ici<=fin) AND (not trouve)) DO
   IF (tab[ici]=valeur_cherchee)
   THEN trouve:=TRUE
   ELSE ici:=ici+1
END;
```

Par exemple, on l'utilise dans le programme suivant :

```
PROGRAM rech_seq;

CONST ind_max = 100;

TYPE
   element = WORD;
   indice = 1..ind_max;
   tableau = ARRAY[indice] of element;

VAR n,i,x: WORD;
     Tabi: tableau;
     ici : indice;
     trouve: boolean;

BEGIN
 WRITELN('Entrez la longueur du tableau a trier : ');
 READLN(n);
 WRITELN('Entrez le nombre a chercher : ');
 READLN(x);
 WRITELN('Entrez les elements du tableau : ');
 FOR  i:=1 TO n DO
    READ(Tabi[i]);
 WRITELN;
 recherche_seq(Tabi,1,n,x);
 IF trouve=TRUE THEN
 WRITELN('Le nombre ',x, ' porte le numero ',ici, ' dans le
     tableau.')
 ELSE WRITELN('Le nombre ',x,' ne se trouve pas dans le tableau.'
     );
 READLN();
 READLN
END.
```

et on obtient :

```
Entrez la longueur du tableau a trier :
4
Entrez le nombre a chercher :
```

```
2
Entrez les elements du tableau :
11
6
2
987

Le nombre 2 porte le numero 3 dans le tableau.
```

3 3 Recherche dichotomique

Pour être plus efficace, divisons pour régner.
Supposons que nous disposions d'un tableau déjà trié dans l'ordre croissant (cf 2.4 page 62).
Considérons un élément situé en son milieu et comparons-le au nombre cherché ce qui orientera notre recherche.

```
PROCEDURE recherche_dich(tab : tableau;
                         debut,fin : indice;
                         valeur_cherchee : element);
  VAR haut,bas : indice;
{-- Si on trouve un element egal a valeur_cherchee alors ici sera
     egal a
  l'indice de cet element et trouve vaudra TRUE.

Parametres : tab (E) -> tableau dans lequel on effectue la
    recherche
             debut (E) -> indice de debut de recherche
             fin (E) -> indice de fin de recherche
             valeur_cherchee (E) -> comme son nom l'indique
             haut,bas (L) -> indice mobiles des sous-tableaux
                 etudies
             trouve (S) -> vrai ou faux selon que la valeur a ete
                 trouvee ou non

Exemple d'appel : recherche_dich(tab,1,10,32) --}

BEGIN
 trouve:=FALSE;
 compteur:=0;
 haut:=fin;
 bas:=debut;
 milieu:=(haut+bas) DIV 2;
 WHILE ((haut>=bas) AND (not trouve)) DO
  BEGIN
   IF (tab[milieu]=valeur_cherchee)
   THEN trouve:=TRUE
   ELSE IF tab[milieu]>valeur_cherchee
```

```
        THEN haut:=milieu-1
        ELSE bas:=milieu+1;
   compteur:=compteur+1;
   milieu:=(bas+haut) DIV 2
  END;
END;
```

On rajoute un compteur et on introduit la procédure dans le programme suivant :

```
BEGIN
 WRITELN('Entrez la longueur du tableau a trier : ');
 READLN(n);
 WRITELN('Entrez le nombre a chercher : ');
 READLN(x);
 WRITELN('Entrez les elements du tableau : ');
 FOR  i:=1 TO n DO
    READ(Tabi[i]);
 WRITELN;
 recherche_dich(Tabi,1,n,x);
 IF trouve=TRUE THEN
 WRITELN('Le nombre ',x, ' porte le numero ',milieu, ' dans le
     tableau.')
 ELSE WRITELN('Le nombre ',x,' ne se trouve pas dans le tableau.'
     );
 WRITELN('En ',compteur,' coups.');
 READLN();
 READLN
END.
```

et on obtient par exemple :

```
Entrez la longueur du tableau a trier :
40
Entrez le nombre a chercher :
21
Entrez les elements du tableau :
0
1
2
3
.
.
.
39

Le nombre 21 porte le numero 22 dans le tableau.
En 4 coups.
```

Quelle est la complexité au pire de cet algorithme ? Comparez-la avec celle de la recherche séquentielle.

Guillaume Connan, Licence Creative Commons BY: $\textcircled{\$}$ $\textcircled{=}$, 2010-2011

4 Algorithmes de tri

4 1 Permutation

Nous aurons, tout au long de cette section, besoin de permuter deux éléments d'un tableau. Nous utiliserons la procédure suivante :

```
PROCEDURE echange(a,b:element);
VAR temp : element;
BEGIN
  temp:=a;
     a:=b;
     b:=temp
END;
```

4 2 Tri par sélection

C'est une méthode élémentaire : elle consiste à déterminer le minimum d'un tableau de nombres, à le placer en tête du tableau résultat et à recommencer sur le tableau initial tronqué.
Commentez le programme suivant :

```
PROGRAM tri_selec2;

 CONST ind_max = 100;

 TYPE
   element = WORD;
   indice = 1..ind_max;
   tableau = ARRAY[indice] of element;

 VAR n,i,compteur: WORD;
     Tabi,tab_trie: tableau;

PROCEDURE  tri_sel(tab:tableau;debut,fin:indice);
VAR i,j,actuel:indice;
    temp:element;
BEGIN
 tab_trie:=tab;
 compteur:=0;
 FOR i:=debut TO fin-1 DO
  BEGIN
   actuel:=i;
     FOR j:=i+1 TO fin DO
       IF tab_trie[j]<tab_trie[actuel]
       THEN  {on echange les valeurs}
         BEGIN
         compteur:=compteur+1;
         temp:=tab_trie[j];
```

```
            tab_trie[j]:=tab_trie[actuel];
            tab_trie[actuel]:=temp
            END;
    END;
END;

BEGIN
 WRITELN('Entrez la longueur du tableau a trier : ');
 READLN(n);
 WRITELN('Entrez les elements du tableau : ');
 FOR  i:=1 TO n DO
    READLN(Tabi[i]);
 WRITELN;
 tri_sel(Tabi,1,n);
 FOR  i:=1 TO n DO
    WRITE(tab_trie[i]:5);
 WRITELN;WRITELN;
 WRITE('En ',compteur,' comparaisons.');
 READLN
END.
```

La fonction **`tri_sel`** effectue n itérations, la i-eme en effectuant $(n-i)$ avec au maximum autant de comparaisons. La complexité au pire, en considérant comme opération élémentaire la comparaison de deux entiers, est donc de l'ordre de $\frac{n(n-1)}{2}$ c'est-à-dire quadratique.

```
Entrez la longueur du tableau a trier :
10
Entrez les elements du tableau :
10
9
8
7
6
5
4
3
2
1

    1    2    3    4    5    6    7    8    9   10

En 45 comparaisons.
```

4 3 Tri par insertion

C'est l'algorithme correspondant à une personne triant des cartes qu'elle saisit une par une sur la table. Nous garderons cette image à l'esprit pour décrire notre algorithme.

Nous balayons un tableau du début à la fin. Au moment d'étudier la carte n°**ind_test**, on prend la **ind_test**e carte et on la met à sa place en la testant avec les carte d'indices inférieurs déjà rangées.

Commentez le programme suivant :

```
PROGRAM tri_inser_it;

 CONST ind_max = 100;

 TYPE
   element = WORD;
   indice = 1..ind_max;
   tableau = ARRAY[indice] of element;

 VAR n,i,compteur: WORD;
     Tab : tableau;

 PROCEDURE  tri_ins_it(VAR tab : tableau; max : indice);
 VAR ind_test, ind_mobil : indice;
     nb_a_inserer : element;
 BEGIN
   FOR ind_test:=2 TO max DO
    BEGIN
      nb_a_inserer:=tab[ind_test];
      ind_mobil:=ind_test-1;
      WHILE (ind_mobil >= 1) AND (tab[ind_mobil] > nb_a_inserer)
          DO
        BEGIN
          tab[ind_mobil+1]:=tab[ind_mobil];
          ind_mobil:=ind_mobil-1;
          compteur:=compteur+1
        END;
      tab[ind_mobil+1]:=nb_a_inserer
    END;
 END;

BEGIN
 WRITELN('Entrez la longueur du tableau a trier :  ');
 READLN(n);
 WRITELN('Entrez les elements du tableau :  ');
 FOR  i:=1 TO n DO
    READLN(Tab[i]);
 tri_ins_it(Tab,n);
 WRITELN;
 FOR  i:=1 TO n DO
    WRITE(Tab[i]:5);
 WRITELN;WRITELN;
 WRITE('En ',compteur,' comparaisons.');
```

```
  READLN
END.
```

Pour information, voici ce que ça peut donner en récursif avec CAML :

```
# let rec insere element = function
   | [] -> [element]
   | tete::queue -> if element <= tete
                    then element::tete::queue
                    else tete::(insere element queue);;
```

```
# let rec tri_insertion = function
   |  [] -> []
   | tete::queue -> insere tete (tri_insertion queue);;
```

Par exemple :

```
# tri_insertion([3;5;2;4;88;75;325;11;21]);;
- : int list = [2; 3; 4; 5; 11; 21; 75; 88; 325]
```

ou encore :

```
# tri_insertion([1.;sqrt(2.);acos(1.);2.*.acos(0.)]);;
- : float list = [0.; 1.; 1.41421356237309515; 3.
   14159265358979312]
```

Bon, pour en revenir à notre procédure **tri_ins_it**, il faudrait déterminer sa complexité.
Nous prendrons cette fois la copie d'un élément dans le tableau comme opération élémentaire.
Au pire, la liste est dans l'ordre inverse. On fait $i-1$ copies pour le i^{e} élément de la liste. Ainsi, le nombre total de copies au pire est :

$$0+1+2+\cdots+n-1=\frac{n(n-1)}{2}$$

donc la complexité au pire est quadratique.

4 4 Tri à bulles

On parcourt le tableau de nombres et on transpose tout couple qui n'est pas dans le bon ordre.
Par exemple, si on veut trier 4-3-2-1, cela donne successivement :
4-3-2-1
3-4-2-1
3-2-4-1
2-3-4-1
2-3-1-4
2-1-3-4
1-2-3-4
Les maxima montent comme des bulles dans un verre d'eau...

Guillaume Connan, Licence Creative Commons , 2010-2011

```
FUNCTION bulle(tab:tableau; nmax : indice):tableau;
VAR tabf: tableau;
    k : indice;
    temp:WORD;
BEGIN
  tabf:=tab;
  k:=1;
  WHILE k < nmax DO
    BEGIN
      IF tabf[k]>tabf[k+1] THEN
        BEGIN
          temp:=tabf[k];
          tabf[k]:=tabf[k+1];
          tabf[k+1]:=temp;
          k:=1
        END
      ELSE k:=k+1
    END;
  bulle:=tabf;
END;
```

On peut également parcourir toute la liste et effectuer les échanges puis recommencer au début de la liste :

```
FUNCTION bulle(tab:tableau; nmax : indice):tableau;
VAR tabf: tableau;
    k : indice;
    temp:WORD;
    echange:BOOLEAN;
BEGIN
  tabf:=tab;
  REPEAT
   echange:=FALSE;
   FOR k:=1 TO nmax-1 DO
    BEGIN
      IF tabf[k]>tabf[k+1] THEN
        BEGIN
          temp:=tabf[k];
          tabf[k]:=tabf[k+1];
          tabf[k+1]:=temp;
          echange:=TRUE;
        END
    END;
  UNTIL echange=FALSE;
  bulle:=tabf;
END;
```

Encore une autre version en partant de la fin :

```
PROGRAM tri_bulle4;

 CONST ind_max = 100;

 TYPE
   element = WORD;
   indice = 1..ind_max;
   tableau = ARRAY[indice] of element;

 VAR n,i,compteur: WORD;
     Tabi,tabf: tableau;

PROCEDURE bulle(tab:tableau; nmin,nmax : indice);
VAR k,j : indice;
    temp:WORD;
BEGIN
  tabf:=tab;
  FOR k:=nmax DOWNTO nmin+1 DO
   FOR j:=nmin TO k-1 DO
    BEGIN
      compteur:=compteur+1;
      IF tabf[j]>tabf[k] THEN
        BEGIN
          temp:=tabf[k];
          tabf[k]:=tabf[j];
          tabf[j]:=temp;
        END
    END;
END;



BEGIN
 WRITELN('Entrez la longueur du tableau a trier : ');
 READLN(n);
 WRITELN('Entrez les elements du tableau : ');
 FOR  i:=1 TO n DO
    READLN(Tabi[i]);
 bulle(Tabi,1,n);
 WRITELN;
 FOR  i:=1 TO n DO
    WRITE(tabf[i]:5);
 WRITELN;WRITELN;
 WRITE('En ',compteur,' comparaisons.');
 READLN
END.
```

Guillaume Connan, Licence Creative Commons, 2010-2011

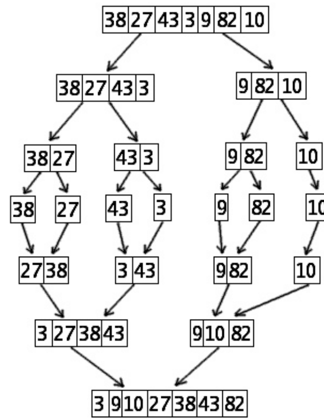
```
Entrez la longueur du tableau a trier :
10
Entrez les elements du tableau :
10
1
2
3
4
5
6
7
8
9

    1    2    3    4    5    6    7    8    9   10

En 45 comparaisons.
```

À chaque itération, on effectue $n-1$ comparaisons. Dans le pire des cas, (le plus petit élément est en dernier), on effectue n itérations : la complexité est donc quadratique ici encore.

Dans le meilleur des cas, la liste est déjà triée et la complexité est linéaire car on n'effectue qu'une itération de $n-1$ comparaisons.

4 5 Tri fusion

On utilise la méthode diviser pour régner : on « coupe » le tableau à trier en deux, on trie chaque tableau et on fusionne les deux tableaux obtenus, de manière récursive. Voici une illustration sur un exemple :



On suppose qu'un type de variable **TAB** a été créé. Commentez cette procédure :

```
PROCEDURE Tri_Fusion (tab : TAB; min, max : INTEGER);
VAR
   mid, i, j, k, ind_min, ind_max: INTEGER;
   tab_aux : TAB;
BEGIN
      IF max > min THEN
      BEGIN
            mid := (min + max) DIV 2;
            Tri_Fusion (tab, min, mid);
            Tri_Fusion (tab, mid + 1, max);

      FOR i := mid DOWNTO min DO
                tab_aux[i] := tab[i];

      FOR j := mid + 1 TO max DO
                tab_aux[mid + 1 + max - j] := tab_aux[j];

      ind_bas := min; ind_haut := max;
            FOR k := min TO max DO
            BEGIN
                  IF tab_aux[ind_bas] < tab_aux[ind_haut] THEN
                  BEGIN
                        tab[k] := tab_aux[ind_bas];
                        ind_bas := ind_bas + 1;
                  END
                  ELSE
                       BEGIN
                              tab[k] := tab_aux[ind_haut];
                              ind_haut := ind_haut - 1;
```

```
            END;
        END;
    END;
END;
```

Pour étudier la complexité de cet algorithme, nous allons prendre comme opération élémentaire la copie d'un élément dans un tableau. Soit n la longueur du tableau à trier. On effectue n premières copies pour créer le tableau auxiliaire puis n nouvelles copies pour modifier le tableau initial. Il faut ensuite ajouter le coût de la fusion des deux sous-tableaux issus des appels récursifs.
Appelons K la fonction coût associé au tri par fusion, alors :

$$\begin{cases} \mathrm{K}(0) = \mathrm{K}(1) = 0 \\ \mathrm{K}(n) = \mathrm{K}\left(\lceil \frac{n}{2} \rceil\right) + \mathrm{K}\left(\lfloor \frac{n}{2} \rfloor\right) + 2n \end{cases}$$

Posons $u_n = \mathrm{K}(n)$. La récurrence définissant la suite (u_n) est compliquée mais, comme pour la dichotomie, nous allons l'étudier dans le cas particulier où n est une puissance de 2. On peut toutefois montrer rapidement par récurrence que la suite (u_n) est croissante.
Posons maintenant $n = 2^k$ et $u_n = u_{2^k} = x_k$, alors $x_0 = 0$ et, pour $k \in \mathbb{N}^*$, $x_k = 2x_{k-1} + 2^{k+1}$. On obtient successivement :

$$x_k = 2\left(2x_{k-2} + 2^k\right) + 2^{k+1} = 2\left(2\left(2x_{k-3} + 2^{k-1}\right) + 2^k\right) + 2^{k+1}$$

On montre alors par récurrence que

$$x_k = 2^k x_0 + k \cdot 2^{k+1} = k \cdot 2^{k+1}$$

Or, pour tout entier n non nul,

$$2^{\lfloor \log_2(n) \rfloor} \leqslant n \leqslant 2^{\lfloor \log_2(n) \rfloor + 1}$$

donc

$$2^{\lfloor \log_2(n) \rfloor + 1} \lfloor \log_2(n) \rfloor \leqslant \mathrm{K}(n) \leqslant 2^{\lfloor \log_2(n) \rfloor + 2} \left(\lfloor \log_2(n) \rfloor + 1\right)$$

c'est-à-dire

$$2n \lfloor \log_2(n) \rfloor \leqslant \mathrm{K}(n) \leqslant 4n\left(\lfloor \log_2(n) \rfloor + 1\right)$$

Finalement

$$\boxed{\mathrm{K}(n) = \Theta\big(n \ln(n)\big)}$$

Ce résultat ne dépend pas du tableau à classer car les opérations sont effectuées systématiquement : la complexité du tri par fusion est donc quasi linéaire dans tous les cas ce qui est une amélioration par rapport aux tris précédents.

4 6 Tri Shell

4 7 Tri rapide

EXERCICES

2 - 1

Écrire une version PASCAL de l'algorithme d'exponentiation rapide en récursif puis en itératif. Prouvez ensuite la terminaison de la version itérative.

2 - 2

Calculez la complexité de l'algorithme suivant en supposant que les $\cdots$ représentent des opérations se déroulant indépendamment de n :

```
Fonction algo( n : entier positif ) : entier
Variable
 | i,k:
Début
 | k ← n
 | TantQue k>0 Faire
 |  | ...
 |  | Pour i variantDe 1 à k Faire
 |  |  | ...
 |  | FinPour
 |  | k ← k DIV 3
 | FinTantQue
 | ...
Fin
```

2 - 3 E3A 2007

Un tableau d'entiers T est dit *trié par ordre croissant* si ses entrées sont triées par ordre croissant : si ℓ est la longueur du tableau, $\forall(i,j) \in \{1,...,\ell\}^2$, $i \leqslant j \Longrightarrow \mathrm{T}[i] \leqslant \mathrm{T}[j]$. Il est dit *trié par ordre décroissant* si ses entrées sont triées par ordre décroissant. Dans les deux cas, il est dit *trié*.
Remarquons qu'un tableau T peut être à la fois trié dans l'ordre croissant et décroissant s'il vérifie : si ℓ est la longueur du tableau, $\forall(i,j) \in \{1,...,\ell\}^2$, $i \leqslant j \Longrightarrow \mathrm{T}[i] = \mathrm{T}[j]$. Il est dit alors *constant*.
Écrire la fonction :

```
Test-tri
    données T : tableau d'entiers
            l : entier
   résultat x : entier
```

qui prend en entrée un tableau T de longueur ℓ et retourne la valeur 1 si le tableau T est trié par ordre croissant et non constant, -1 si T est trié par ordre décroissant et non constant, 0 si T est constant et 2 si T n'est pas trié.
Écrire la procédure :

```
Fusion
    données T : tableau d'entiers
            l : entier
            U : tableau d'entiers
            m : entier
   résultat W : tableau d'entiers trié
```

qui fusionne deux tableaux triés tous deux par ordre croissant (respectivement tous deux triés par ordre décroissant), de longueurs respectives ℓ, m ; le résultat est un tableau trié par ordre croissant (respectivement décroissant) de longueur $\ell + m$. Dans le cas où les deux tableaux ne sont pas triés dans un même ordre, la procédure **Fusion** renverra un tableau vide et un message d'avertissement.

2 - 4 Centrale 2007 : Autour de la suite de Fibonacci

L'objectif de ce problème est de comparer les complexités temporelles (et dans une moindre mesure spatiale) d'algorithmes de calcul numérique de suites entières vérifiant une relation de récurrence linéaire. Les entiers manipulés pourront être arbitrairement long (ce qui n'est a priori pas le cas ni en Caml ni en Pascal). Dans les calculs de complexité, on demande des ordres de grandeur du nombre d'opérations élémentaires sur les bits.
Exemple 1 : l'algorithme naturel pour sommer deux entiers de n bits demande $\mathrm{O}(n)$ opérations élémentaires (additions bit-à-bit, propagation de retenue).
On ne demande pas d'équivalents, mais des majorants asymptotiques.
Lorsque $u_n = \mathrm{O}(v_n)$ et $v_n = \mathrm{O}(u_n)$, on pourra noter $u_n = \Theta(v_n)$, ce qui est plus précis que $u_n = \mathrm{O}(v_n)$.

Exemple 2 : le « tri bulle » d'un tableau de n valeurs effectue $\Theta(n^2)$ comparaisons de valeurs, et réalise$O(n^2)$ échanges dans le tableau.
Les candidats rédigeant en Caml devront donner le type de chaque fonction écrite.
Notation : $a[b]$ désignera a modulo b, c'est à dire le reste dans la division euclidienne de a par b.

1. a. **Questions préliminaires.**
   b. Évaluer le nombre d'opérations élémentaires sur des bits requises pour effectuer le produit de deux entiers de n bits avec une méthode élémentaire (que l'on décrira sommairement).
   c. Donner un algorithme effectuant une multiplication d'entiers avec une meilleure complexité. Le décrire, et donner (sans justification) sa complexité temporelle.
   d. Décrire très sommairement l'algorithme d'exponentiation rapide. Justifier son intérêt par rapport à un algorithme élémentaire.
2. **Diverses façons de calculer f_n** .
   La suite de Fibonacci est définie par les relations $f_0 = 0$, $f_1 = 1$, et pour tout $n \in \mathbb{N}$:

$$f_{n+2} = f_n + f_{n+1}$$

   a. En utilisant les résultats standards sur les suites vérifiant une relation de récurrence linéaire d'ordre 2, à coefficients constants, donner la valeur de f_n , et l'ordre de grandeur de sa représentation binaire (écriture en base 2). En déduire un minorant pour le temps de calcul de tout programme calculant f_n.
   b. Écrire une fonction récursive `fibo` prenant en entrée un entier n et retournant f_n en appliquant directement la définition de f.
   c. Prouver que le nombre d'appels récursifs effectués pour calculer f_n avec la fonction précédente est exponentiel en n.
   d. Écrire une nouvelle fonction `fibo2` réalisant le calcul de f_n de façon itérative, en calculant de proche en proche les f_k pour $k \leqslant n$. Donner le nombre d'additions d'entiers qui seront effectuées lors du calcul de f_n, et évaluer la dépendance du temps de calcul par rapport à n, ainsi que la place en mémoire requise (on supposera que les entiers calculés restent inférieurs au plus grand entier représentable en Caml ou en Pascal).
   e. On observe qu'en notant $X_n = \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix}$ et $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, on a $X_{n+1} = AX_n$ pour tout $n \in \mathbb{N}$, puis $X_n = A^n X_0$. Estimer le temps de calcul et la place en mémoire requise pour calculer f_n en exploitant cette relation.
   f. Si on cherche à calculer f_n modulo un entier k « petit » (dans un sens à préciser par le candidat), quelle méthode peut-on utiliser, et quels sont le temps et l'espace requis pas cette méthode ?

2 - 5 École de l'Air

On dispose d'une pile de rondelles de diamètres variés, que l'on souhaite ranger par ordre décroissant de taille, la plus large devant être en bas de la pile. Malheureusement, on ne peut manipuler la pile que « par morceaux » : la seule opération autorisée consiste à prendre la partie de la pile surmontant une certaine rondelle et à retourner d'un bloc tout le paquet.
Par exemple, si une pile contient des rondelles de diamètres 7, 10, 20, 2 et 5 et que l'on retourne le bloc commençant à la rondelle de diamètre 20, on obtient la pile de diamètres successifs 7, 10, 5, 2 et 20.

Nous modéliserons la pile de rondelles par un tableau d d'entiers, indicé de 0 à $n-1$, où n est le nombre de rondelles, la case numéro i contenant le diamètre en millimètres de la rondelle i. La case numéro 0 correspond au bas de la pile.
Nous supposerons la constante n définie.
En Pascal, d est de type `Pile` défini par :

```
Pile = ARRAY[0..n-1] OF INTEGER
```

1. Écrire une procédure

retourne(VAR d:Pile ; i:INTEGER)
telle que **retourne(d,i)** modifie le tableau **d** pour « retourner » la partie supérieure de la pile à partir de la rondelle d'indice i (incluse).

2. Quel est, en nombre d'affectations dans **d**, le coût de ce retournement ?

3. Comment faire pour mettre en bas de la pile la rondelle la plus large ?

4. Écrire une procédure
place(VAR d:Pile ; i:INTEGER)
telle que **place(d,i)** modifie le tableau **d** pour mettre l'indice i la rondelle la plus large de la partie de la pile surmontant la rondelle d'indice i (incluse) ;

5. En déduire une fonction ou une procédure qui trie **d**.

6. Quel est le coût au pire de cette méthode, en nombre de retournements d'un morceau de pile ?

Guillaume Connan, Licence Creative Commons BY: $-$, 2010-2011

DES SOLUTIONS

2 - 3 E3A 2007

```
   PROGRAM e3a2007;

 CONST ind_max = 100;

 TYPE
   element = SHORTINT;
   indice = 1..ind_max;
   tableau = ARRAY[indice] of element;

 VAR n,i: WORD;
     Tabi: tableau;


FUNCTION  test_tri(T:tableau;l:WORD):SHORTINT;
VAR k,j:indice;
    delta:element;
BEGIN
 k:=1;
 WHILE ((T[k+1]=T[k]) AND (k<=l-2)) DO k:=k+1;
 IF ((k=l-1) AND (T[k+1]=T[k])) THEN test_tri:
     =0
 ELSE BEGIN
  j:=k;
  delta:=(T[j+1]-T[j]);
  WHILE ((delta*(T[j+1]-T[j])>=0) AND (j<=l-2)
      ) DO j:=j+1;
  IF ((j=l-1) AND (delta*(T[j+1]-T[j])>=0))
      THEN BEGIN
   IF delta>0 THEN test_tri:=1
   ELSE test_tri:=-1
  END
  ELSE test_tri:=2
 END;

END;


{--Traitement de la procedure par Pascal--}

BEGIN
 WRITELN('Entrez la longueur du tableau a
      tester : ');
 READLN(n);
 WRITELN('Entrez les elements du tableau : ');
 FOR  i:=1 TO n DO
    READLN(Tabi[i]);
 WRITELN;
 WRITE(test_tri(Tabi,n));
 READLN
END.
```

```
Entrez la longueur du tableau a tester :
3
Entrez les elements du tableau :
8
8
9

1
```

Voici une version pour la question 2 sans le test de croissance par gain de place (rajouter un IF test_tri(T)=1 AND test_tri(U)=1) :

```
PROGRAM e3a2007_2;

 CONST ind_max = 100;

 TYPE
   element = LONGINT;
   indice = 1..ind_max;
   tableau = ARRAY[indice] of element;

 VAR n1,n2,i: WORD;
     W,Tab1,Tab2: tableau;


PROCEDURE  Fusion(T,U:tableau;l,m:WORD);
VAR kmin,kmax,k:indice;
    lmin,lmax,j:WORD;
    Tmin,Tmax:tableau;
BEGIN
 k:=1;
 kmin:=1;
 kmax:=1;
 {--Recherche de l element maxi--}
 IF T[l]<=U[m] THEN
   BEGIN
    lmin:=l; Tmin:=T; lmax:=m; Tmax:=U
   END
 ELSE
   BEGIN
    lmin:=m; Tmin:=U; lmax:=l; Tmax:=T
   END;
 {--tri des deux tableaux--}
 WHILE kmin<=lmin DO
   BEGIN
    IF Tmin[kmin]<=Tmax[kmax] THEN
      BEGIN
       W[k]:=Tmin[kmin]; k:=k+1; kmin:=kmin+1
      END
    ELSE
      BEGIN
       W[k]:=Tmax[kmax]; k:=k+1; kmax:=kmax+1
      END
```

```
   END;
  {--Rajout des derniers elements du tableau
      contenant le max--}
  FOR j:=0 TO (lmax-kmax) DO W[k+j]:=Tmax[kmax+
      j]
END;

{--Traitement de la procedure par Pascal--}

BEGIN
 WRITELN('Entrez la longueur du premier
      tableau a fusionner : ');
 READLN(n1);
 WRITELN('Entrez les elements du premier
      tableau : ');
 FOR  i:=1 TO n1 DO
    READLN(Tab1[i]);
 WRITELN;
 WRITELN('Entrez la longueur du deuxieme
      tableau a fusionner : ');
 READLN(n2);
 WRITELN('Entrez les elements du deuxieme
      tableau : ');
 FOR  i:=1 TO n2 DO
    READLN(Tab2[i]);
 WRITELN;
 Fusion(Tab1,Tab2,n1,n2);
 FOR  i:=1 TO n1+n2-1 DO
 WRITE(W[i],' , ');
 WRITE(W[n1+n2]);
 READLN
END.
```

Par exemple :

```
Entrez la longueur du premier tableau
a fusionner :
5
Entrez les elements du premier tableau :

3
66
77
1000

Entrez la longueur du deuxieme tableau
a fusionner :
8
Entrez les elements du deuxieme tableau :
-2
0
1
2
4
5
888
999
-2,0,0,1,2,3,4,5,66,77,888,999,1000
```

2 - 5 École de l'Air

1. Faisons un petit tableau pour comprendre :

$n-1$	$n-1-1$	$n-1-2$	...	$n-1-(n-i-2)$	$n-1-(n-i-1)$
i	$i+1$	$i+2$	...	$i+(n-i-2)$	$i+(n-i-1)$

On comprend donc que `d[i+k]` doit devenir `d[n-1-k]`.

```
FUNCTION retourne(d:Pile ; i:INTEGER):
     Pile;
VAR dd:Pile;
      k:INTEGER;
BEGIN
 dd:=d;
 FOR k:=0 TO n-i-1 DO dd[i+k]:=d[n-1-k]
 retourne:=dd
END;
```

2. Il y a eu une affectation pour créer **dd** que l'on peut négliger puis $n-i$ pour les échanges.

3. On commence par créer une fonction qui trouve le rang de la rondelle de taille maximum.

```
FUNCTION ind_maxi(d:Pile ; i:INTEGER):
     INTEGER;
VAR ind_max,j:INTEGER;
BEGIN
  ind_max:=i;
  FOR j:=i+1 TO n-1 DO
    IF d[j]>d[ind_max] THEN ind_max:=j
  ind_maxi:=ind_max
END;
```

puis on retourne la pile à partir de **ind_maxi** pour mettre la plus grosse rondelle en dernieravec **retourne(d,ind_max)** puis on retourne totalement la pile pour mettre la plus grosse rondelle en premier avec **retourne(d,0)**.

4. On utilise la même idée que précédemment :

```
PROCEDURE place(VAR d:Pile ; i:INTEGER);
VAR ind_max:INTEGER;

BEGIN
  ind_max:=ind_maxi(d,i);
```

```
  retourne(d,ind_max);
  retourne(d,i);
END;
```

5. On utilise successivement la procédure `place` de bas en haut pour placer dans l'ordre décroissant les rondelles :

```
PROCEDURE tri(d:Pile);
VAR i:INTEGER;
BEGIN
  FOR i:=0 TO n-2 DO
    place(d,i);
END;
```

6. La question posée est sans intérêt : chaque appel à `place` effectue deux retournements et il y a $n-1$ appels à place donc il y a $2(n-1)$ retournements.

La procédure `place(d,i)` appelle trois procédures qui effectuent des affectations :

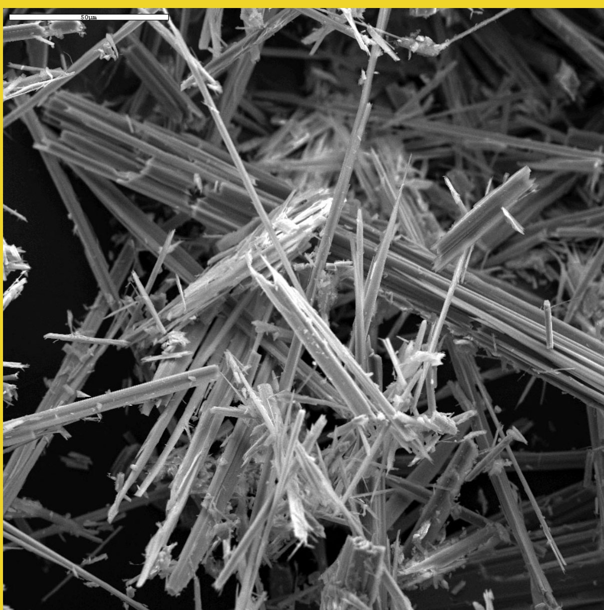
- `ind_maxi` qui effectue au pire `n-i` affectations ;
- `retourne(d,ind_maxi)` qui effectue `n-ind_maxi` donc au pire `n-i` affectations ;
- `retourne(d,i)` qui effectue `n-i` affectations.

Donc au pire, il y aura $\sum_{i=0}^{n-1} 3(n-i)$ affectations c'est-à-dire $\frac{3}{2}n(n+1)$: la complexité au pire est quadratique en nombre d'affectations.

OPTION

3

Structures de données I



1 Enregistrements

1 1 Une classe de rationnels

Avant de rentrer dans le vif du sujet, nous allons nous familiariser un peu avec la notion d'enregistrement (RECORD en Pascal dont on a parlé à la section 1.1.1.8 page 34) qui s'apparente un peu à la notion de *classe* de la programmation orientée objet (POO).
Nous allons donc créer une classe d'objet assez familiers mathématiquement parlant : les rationnels. Sur Pascal, on crée un nouveau TYPE :

```
TYPE
 Rationnel=RECORD
  Num, Den : INT64;
  END;
```

Si r est de type Rationnel, alors r.Num sera son dénominateur de type INT64 et r.Den son dénominateur.
On peut alors imaginer une procédure qui lit un rationnel entré par l'utilisateur. On découvre la procédure HALT qui permet d'interrompre un programme en cours d'exécution et possède un argument facultatif (du type WORD) correspondant à un code de sortie.

```
PROCEDURE EntrerR(VAR r:Rationnel);
BEGIN
 WRITELN('Vous allez entrer un rationnel');
 WRITE('Numerateur   : '); READLN(r.Num);
 WRITE('Denominateur : '); READLN(r.Den);
 IF r.Den=0 THEN
  BEGIN
   WRITELN('Le denominateur doit etre non nul !.');
   HALT;
  END;
END;
```

On va essayer de faire un peu plus moderne (si, si, c'est possible, même avec Pascal...) en créant une procédure qui nous permette d'écrire un rationnel sous la forme « a/b » en faisant de la manipulation de chaînes de caractères à l'aide des fonctions POS, COPY, VAL (voir la section 1.1.1.6 page 33).

```
PROCEDURE EntrerR2(VAR rat:Rationnel);
VAR
 R,N,D:STRING;
 p,code_err:WORD;
BEGIN
 READLN(R);{on entre le nombre sous la forme a/b }
 p:=POS('/',R); { on repere le / }
 IF p<>0 THEN { si ce / existe bien }
```

```
  BEGIN
   D:=COPY(R,p+1,LENGTH(R)-p); { on repere le denominateur }
   N:=COPY(R,1,p-1); { et le numerateur }
  END
 ELSE
   BEGIN
     N:=R;
     D:='' { on peut entrer un entier sans / }
   END;
 VAL(N,rat.Num,code_err); {on transforme la chaine N en nombre}
 IF (code_err=0) THEN {si la conversion s''est bien passee}
  IF D<>'' THEN VAL(D,rat.Den,code_err) ELSE rat.Den:=1;
  {on distingue les entiers des non entiers}
 IF code_err<>0 THEN {si l'une des conversions s'est mal passee}
  BEGIN
   WRITELN('Erreur d''ecriture');
   READLN;
   HALT
  END;
 IF rat.Den=0 THEN {on ne divise pas par zero !}
  BEGIN
   WRITELN('Denominateur nul');
   READLN;
   HALT
  END;
END;
```

On demande aussi à Pascal d'afficher le rationnel de manière conviviale :

```
PROCEDURE Afficher(r:Rationnel);
BEGIN
 WRITELN(r.Num,'/',r.Den);
END;
```

Maintenant, à vous de jouer : créez des fonctions qui simplifient, additionnent, soustraient, multiplient, divisent, élèvent à une puissance entière des rationnels.

1 2 Méthode de Héron

Nous avons déjà étudié l'algorithme de HÉRON page 26. Utilisez-le pour obtenir des approximations rationnelles d'irrationnels à l'aide des outils précédemment introduits.

1 3 Fractions continues

– Vous connaissez l'algorithme suivant :

$$\begin{aligned} 172 &= 3\,51 + 19 \\ 51 &= 2\,19 + 13 \\ 19 &= 1\,13 + 6 \\ 13 &= 2\,6 + 1 \\ 6 &= 6\,1 + 0 \end{aligned}$$

– On peut donc facilement compléter la suite d'égalité suivante :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = \ldots$$

Pratique du développement

– Quand tous les numérateurs sont égaux à 1, on dit qu'on a développé $\frac{172}{51}$ en fraction continue, et pour simplifier l'écriture on note :

$$\frac{172}{51} = [3, 2, \ldots]$$

– Par exemple, on peut développer $\frac{453}{54}$ en fraction continue.
– Dans l'autre sens on peut écrire $[2, 5, 4]$ sous la forme d'une fraction irréductible.

Nous voudrions utiliser Pascal pour obtenir de tels développements mais il nous faudrait pouvoir travailler sur des listes qui seront notre premier exemple de *structures ordonnées* mais avant, il nous faut découvrir un nouvel outil...

2 Pointeurs et pointés

Une fois n'est pas coutume, nous allons parler un peu technique. Quand vous déclarer une variable sur Pascal, elle est forcément d'un certain type. Selon ce type, Pascal va réserver un espace mémoire d'une certaine taille. Or le compilateur ne va pas accepter des variables lui demandant trop d'espace mémoire. Une astuce est donc de passer par un *pointeur* qui est une variable qui contient l'adresse mémoire d'une autre variable déjà stockée. Ainsi, à la compilation, un pointeur ne fera pas grincer la machine car la taille d'un pointeur est de quatre octets, quelque soit la variable pointé.
C'est un peu comme quand vous écrivez l'adresse de votre maison sur un bout de papier : vous pouvez ainsi, en quelque sorte, mettre votre maison dans une enveloppe sans que le facteur n'ai a porter plusieurs tonnes de parpaings...
Cela permet une gestion *dynamique* de la mémoire : par exemple, jusqu'à maintenant, on a utilisé des sortes de listes sous forme de tableau dont il fallait déclarer à l'avance la taille, ce qui correspond à une gestion *statique* de la mémoire et conduit à une mauvaise utilisation de la mémoire.

Par exemple, si vous cherchez à définir la liste des diviseurs d'un entier, vous ne savez pas a priori quelle sera sa taille : celle-ci varie dynamiquement au cours de l'exécution du programme.
Avec les pointeurs, seuls ceux-ci se voient allouer de la mémoire (et très peu : 4 octets) pour toute la durée de l'exécution du programme.
Nommons commodément p le pointeur. La variable pointée sera alors désignée par p^. Pour pouvoir l'utiliser, il faut lui réserver dynamiquement de la mémoire en tapant NEW(p). Dès qu'on n'en a plus besoin, on peut libérer la place en mémoire en tapant DISPOSE(p).
On peut donc allouer et vider de la mémoire à une variable pointée par un même pointeur.
On définit deux pointeurs. Que donne les quatre séquences d'instructions suivantes :

```
VAR a,b : ^INTEGER;

BEGIN
NEW(a); a^:=1; b:=a; WRITELN(b^);
NEW(a);NEW(b); a^:=1; b:=a; WRITELN(b^);
NEW(a); a^:=2; b:=a; DISPOSE(a); WRITELN(b^);
NEW(a); a^:=1; b:=a; b^:=2; WRITELN(a^);
READLN
END.
```

3 Listes chaînées définies récursivement

3 1 Qu'est-ce que c'est ?

Récursivement, une liste c'est tout simple : c'est soit le vide, soit un élément suivi d'une liste, ce qui s'écrit de manière standard :

```
Liste = Nil + Élément x Liste
```

le x renvoyant au produit cartésien et Nil (« *Not In List* ») représentatnt une liste vide.

3 2 Comment ?

Une liste sera représentée par un pointeur vers un premier chaînon constitué d'un élément (la tête) et d'une liste (la queue) :

```
TYPE
   Element = INTEGER;
   Liste = ^Chainon;

   Chainon = RECORD
     tete  : Element;
     queue : Liste;
   END;
```

Guillaume Connan, Licence Creative Commons, 2010-2011

Cette construction correspond bien à la définition récursive donnée précédemment.

3 3 Outils de base

Il est temps de construire quelques outils qui nous seront fort utiles par la suite.

```
FUNCTION Queue(L:Liste):Liste;
BEGIN
 Queue:=L^.queue;
END;


FUNCTION Tete(L:Liste):Element;
BEGIN
 Tete:=L^.tete;
END;


FUNCTION Est_Vide(L:Liste):BOOLEAN;
BEGIN
 Est_Vide:=(L=NIL);
END;
```

Ensuite, on veut ajouter un élément à une liste « par la tête » :

```
FUNCTION Colle_en_tete(e:Element; L:Liste):Liste;
VAR L2:Liste;
BEGIN
 NEW(L2);
 L2^.tete:=e;
 L2^.queue:=L;
 Colle_en_tete:=L2;
END;
```

Pour concaténer deux listes, la première étant à droite :

```
FUNCTION Concat(L1,L2:Liste):Liste;
  BEGIN
    IF L1 = NIL THEN
     Concat:=L2
    ELSE
     Concat:=Colle_en_tete(L1^.tete,Concat(L1^.queue,L2))
  END;
```

Pour ajouter un élément « par la queue » :

```
FUNCTION Colle_en_queue(e:Element; L:Liste):Liste;
VAR L2:Liste;
BEGIN
 NEW(L2);
```

```
 L2^.tete:=e;
 L2^.queue:=NIL;
 Colle_en_queue:=Concat(L,L2)
END;
```

On aura également besoin de créer des listes :

```
FUNCTION Cree:Liste;
{ on fait rentrer les valeurs par la gauche }
VAR
 L:Liste;
 e:Element;
BEGIN
 L:=NIL;
 REPEAT
  WRITE('Rentrer une valeur : '); READLN(e);
  IF e>=0 THEN L:=Colle_en_queue(e,L);
 UNTIL e<0;
 Cree:=L;
END;
```

On veut pouvoir afficher les listes. Par commodité ici, on ne travaillera qu'avec des entiers positifs : le signal de fin de liste sera donc l'entrée d'un nombre négatif... Voici comment afficher les listes de la gauche vers la droite :

```
PROCEDURE Affiche(L:Liste);
BEGIN
 IF Est_Vide(L) THEN WRITELN
 ELSE
  BEGIN
   WRITE(Tete(L):5); { pour espacer les elements }
   Affiche(Queue(L));
  END;
END;
```

Pour calculer leur longueur :

```
FUNCTION Longueur(L:Liste):Integer;
BEGIN
 IF Est_Vide(L) THEN Longueur:=0
 ELSE Longueur:=1+Longueur(Queue(L));
END;
```

Pour extraire le i-eme élément d'une liste en commençant à compter à partir de 1 :

```
FUNCTION ieme(L:Liste;pos:INTEGER):Element;
BEGIN
  IF pos=1 THEN ieme:= L^.tete
  ELSE ieme:=ieme(L^.queue,pos-1)
END;
```

Testons en vrai :

```
VAR L1,L2,L:Liste;
    i : INTEGER;

BEGIN
 L1:=Cree;
 Affiche(L1);
 WRITELN;
 WRITELN(' La longueur de la liste est : ',Longueur(L1));
 L2:=Cree;
 Affiche(L2);
 WRITELN;
 WRITELN(' La longueur de la liste est : ',Longueur(L2));
 L:=Concat(L1,L2);
 WRITELN(' La liste concatenee est : ');
 Affiche(L);
 WRITELN;
 WRITELN(' La longueur de la liste concatenee est : ',Longueur(L)
     );
 WRITELN(' Entrez un entier positif ');
 READLN(i);
 IF i<= Longueur(Concat(L1,L2)) THEN
    WRITELN('Le ',i,'eme element de la liste concatenee est : ',
        ieme(L,i))
 ELSE
    WRITELN('Cet entier est trop grand');
 WRITE('Appuyer sur Entree pour terminer.');
 READLN;
END.
```

Par exemple :

```
Rentrer une valeur : 1
Rentrer une valeur : 2
Rentrer une valeur : 3
Rentrer une valeur : 4
Rentrer une valeur : 5
Rentrer une valeur : 6
Rentrer une valeur : -1
    1    2    3    4    5    6

 La longueur de la liste est : 6
Rentrer une valeur : 7
Rentrer une valeur : 8
Rentrer une valeur : 9
Rentrer une valeur : -1
    7    8    9
```

```
 La longueur de la liste est : 3
 La liste concatenee est :
    1    2    3    4    5    6    7    8    9

 La longueur de la liste concatenee est : 9
 Entrez un entier positif
3
Le 3eme element de la liste concatenee est : 3
Appuyer sur Entree pour terminer.
```

3 4 Fractions continues

Revenons à notre problème de développement de rationnels en fractions continues (cf section 3.1.3 page 79).

```
FUNCTION inverseFC(r:Rationnel):Rationnel;
  VAR ir:Rationnel;
  BEGIN
  ir.Num:=r.Den;
  ir.Den:=r.Num MOD r.Den;
  inverseFC:= ir
  END;


FUNCTION fc(r:Rationnel):Liste;
 BEGIN
  IF r.Den=0 THEN
    fc:= NIL
  ELSE
    fc:=Colle_en_tete(r.Num DIV r.Den , fc(inverseFC(r)))
 END;

VAR r:Rationnel;

BEGIN
 WRITE('Entrer un rationnel : '); EntrerR2(r);
 WRITELN;
 WRITELN('Le developpement en fraction continue de ',r.Num,'/',r.
     Den ,' est : ');
 Affiche(fc(r));
 WRITE('Appuyer sur Entree pour terminer.');
 READLN
END.
```

Ce qui donne :

```
Entrer un rationnel : 172/51
Le developpement en fraction continue de 172/51 est :
    3    2    1    2    6
Appuyer sur Entree pour terminer.
```

4 Listes chaînées définies itérativement

4 1 Principe

Cette fois, on définit une liste à partir de son premier élément, de son nombre d'éléments et de pointeur qui permet de progresser d'un élément au suivant. C'est plus lourd à utiliser.

```
TYPE
  Elem = INTEGER;
  Pointeur = ^Maillon;
  Maillon = RECORD
    element : Elem;
    suivant : Pointeur;
    END;
  Liste = RECORD
    taille : WORD;
    premier : Pointeur;
  END;
```

4 2 Outils de base

Comme pour le type récursif, on va construire un certain nombre d'outils de base.

4 2 1 Liste vide

```
PROCEDURE
```

5 Piles

5 1 Qu'est-ce que c'est ?

C'est une liste mais qui a des accès limités comme une pile d'assiettes : on ne peut y accéder que par le sommet, que ce soit pour ajouter ou supprimer une assiette ou regarder son motif.
On utilisera le même type que pour les listes définies récursivement.
On n'utilisera que les fonctions :

- `Empiler(e:Element ; p:Pile):Pile` qui ajoute une assiette ;
- `Depiler(p:Pile):Piles;)` qui enlève l'assiette du sommet ;
- `Sommet(p:Pile):Element;)` qui regarde le motif de l'assiette visible au sommet.

On définira un type `Pile` adapté du type `List` mais en changeant les noms :

```
TYPE
Element = STRING;
Pile = ^Assiette;
```

```
Assiette = RECORD
  motif : Element;
  dessous : Pile;
END;
```

5 2 Outils

Nous aurons besoin de construire...deux constructeurs :

- une pile vide :

```
FUNCTION PileVide : Pile;
BEGIN
 PileVide:=NIL
END;
```

- une pile en ajoutant un élément à une pile déjà existante :

```
FUNCTION  Empiler(e:Element ; p:Pile):Pile;
VAR plate : Pile;
BEGIN
 NEW(plate);
 plate^.motif := e;
 plate^.dessous := p;
 Empiler:=plate
END;
```

On aura besoin d'un prédicat testant si une liste est vide :

```
FUNCTION EstVide(p:Pile):BOOLEAN;
BEGIN
 EstVide:=(p=NIL)
END;
```

Une fonction retournant le sommet de la pile :

```
FUNCTION Sommet(p:Pile):Element;
BEGIN
 IF p=NIL THEN {On traite l'erreur}
   BEGIN
     WRITELN('Pile vide');
     HALT
   END
 ELSE Sommet:=p^.motif
END;
```

Une fonction enlevant le sommet :

```
FUNCTION Depiler(p:Pile):Pile;
BEGIN
 Depiler:=p^.dessous;
```

```
END;
```

Pour le traitement purement « Pascalien », on aura besoin d'une procédure affichant une pile :

```
PROCEDURE Affiche(p:Pile);
VAR ptemp:Pile;
BEGIN
 ptemp:=p;
 WHILE  NOT EstVide(ptemp) DO
  BEGIN
   WRITELN(Sommet(ptemp));
   ptemp:=Depiler(ptemp);
  END;
END;
```

et créant une pile (attention à l'ordre !) :

```
FUNCTION Cree:Pile;
VAR
 P:Pile;
 e:Element;
BEGIN
 P:=PileVide;
 WHILE e<>'Stop' DO
  BEGIN
  WRITE('Rentrer une valeur ou un operateur :  '); READLN(e);
  P:=Empiler(e,P);
  END;
  Cree:=Depiler(P)
END;
```

5 3 Évaluation d'une expression arithmétique postfixée

Commençons par un cas simple : on multiplie ou additionne des nombres positifs et nous allons implémenter ces opérations de manière linéaire donc sans utiliser de parenthèses.
Par exemple, 5 4 3 + * donne 35.
Comme les nombres et les opérateurs sont apparemment de types différents, on va utiliser des chaînes de caractères.
On lit maintenant une pile et on empile les valeurs en effectuant des opérations sur les deux derniers nombres empilés dès qu'arrive en tête de pile une opération binaire et on empile le résultat.

```
FUNCTION polonais(pol:Pile):Pile;
VAR pile_nb,pile_tmp:Pile;
    nb,nb1,nb2,indic : Integer;
    unite,ch : STRING;

BEGIN
```

```
 pile_nb:=PileVide;
 pile_tmp:=pol;
 WHILE (NOT EstVide(pile_tmp)) DO
  BEGIN
   unite:=Sommet(pile_tmp);
   VAL(unite,nb,indic);
   CASE indic OF
     0 : pile_nb:=Empiler(unite,pile_nb)
     ELSE
      BEGIN
        VAL(Sommet(pile_nb),nb2,indic);
        pile_nb:=Depiler(pile_nb);
        VAL(Sommet(pile_nb),nb1,indic);
        pile_nb:=Depiler(pile_nb);
        CASE unite[1] OF
          '+' : BEGIN
                 STR(nb1+nb2,ch);
                 pile_nb:=Empiler(ch,pile_nb);
                END;
          '*' : BEGIN
                 STR(nb1*nb2,ch);
                 pile_nb:=Empiler(ch,pile_nb);
                END;
        END;
      END;
   END;
   pile_tmp:=Depiler(pile_tmp);
  END;
polonais:=pile_nb;
END;
```

Par exemple, en reprenant l'exemple donné plus haut :

```
VAR P,Pol:Pile;

BEGIN
P:=Cree;
Affiche(P);
READLN;
WRITELN('Le resultat de l''operation est : ');
Pol:=polonais(P);
Affiche(Pol);
READLN
END.
```

donne :

```
Rentrer une valeur ou un operateur : *
Rentrer une valeur ou un operateur : +
```

```
Rentrer une valeur ou un operateur : 3
Rentrer une valeur ou un operateur : 4
Rentrer une valeur ou un operateur : 5
Rentrer une valeur ou un operateur : Stop
5
4
3
+
*

Le resultat de l'operation est :
35
```

EXERCICES

3 - 1 Manipulations de listes chaînées définies récursivement

1. Déterminer :

```
Retourne(L:Liste):Liste;
```

qui écrit une liste à l'envers.

2. Déterminer :

```
Appartient(e:Element;L:Liste):BOOLEAN;
```

qui vérifie si un élément appartient à une liste.

3. Déterminer :

```
Rechercher(e:Element;L:Liste):WORD;
```

qui détermine le rang de la première apparition d'un élément dans une liste. Renvoie -1 si l'élément n'appartient pas à la liste.

4. Déterminer :

```
Extraire(d,f:WORD; L:Liste):Liste;
```

qui extrait la sous-liste des éléments de `L` compris entre les rangs `d` et `f` compris.

5. Déterminer :

```
InsererMilieu(e:Element;pos:WORD;L:Liste)
    :Liste;
```

qui insère un élément `e` à la position `pos` dans la liste `L`.

6. Déterminer :

```
Filtre(L:Liste):Liste;
```

qui filtre les éléments d'une liste selon un certain test. Par exemple, on écrira le filtre pour le test `.>3`.

7. Déterminer :

```
Supprimer(pos:WORD ; L:Liste):Liste;
```

qui supprime l'élément de rang `pos` de la liste `L`.

3 - 2 Liste des diviseurs

Déterminer une fonction qui donne la liste des diviseurs d'un entier. On gardera en mémoire que les diviseurs vont souvent par deux...

3 - 3 Liste des nombres parfaits

Un nombre parfait est un nombre égal à la somme de ses diviseurs (distincts de lui-même...).
Déterminer une fonction qui donne la liste des nombres parfaits inférieurs à un entier n donné.

3 - 4 Crible d'Ératosthène récursif

1. Créer une fonction récursive qui écrit la liste des entiers compris entre les entiers a et b.
2. Créer une fonction récursive qui retire les multiples d'un entier n dans une liste L.
3. Créer finalement une fonction récursive qui dresse la liste des entiers premiers inférieurs à n selon le principe du crible d'Ératosthène.

3 - 5 Décomposition en facteurs premiers récursive

En utilisant **`crible`**, créer une fonction qui renvoie la liste des facteurs de sa décomposition en produit de facteurs premiers.

3 - 6 Changement de base récursif

Déterminer une fonction qui donne la liste des chiffres de l'écriture en base b (avec $2 \leqslant b \leqslant 10$) d'un entier n écrit en base 10.

3 - 7 Épreuve de l'X, 2004

Il s'agit de chercher le premier problème intitulé *Sélection*. Pour le deuxième, il faudra attendre l'an prochain et l'étude des arbres. http ://www.imprimerie.polytechnique.fr/EnLignes/Files/04_Info_MPInfo_4h.pdf

Guillaume Connan, Licence Creative Commons BY: $\circledS$ $\circledcirc$, 2010-2011

DES SOLUTIONS

3 - 1 Manipulations de listes chaînées définies récursivement

1. Une solution :

```
FUNCTION Retourne(L:Liste):Liste;
BEGIN
  IF Est_Vide(L) THEN
     Retourne:=NIL
  ELSE
     Retourne:=Concat(Retourne(Queue(L)),
          Colle_en_tete(Tete(L),NIL))
END;
```

et une autre :

```
FUNCTION Retourne2(L:Liste):Liste;
BEGIN
  IF Est_Vide(L) THEN
     Retourne2:=NIL
  ELSE
     Retourne2:=Colle_en_queue(Tete(L),
          Retourne2(Queue(L)))
END;
```

2. Testons si un élément appartient à une liste récursivement :

```
FUNCTION Appartient(e:Element;L:Liste):
     BOOLEAN;
BEGIN
 IF Est_Vide(L) THEN Appartient:=FALSE
 ELSE IF Tete(L)=e THEN Appartient:=TRUE
      ELSE Appartient:=Appartient(e,Queue
          (L))
END;
```

3. On utilise `Appartient` :

```
FUNCTION Rechercher(e:Element;L:Liste):
     SHORTINT;
BEGIN
 IF Appartient(e,L) THEN
     IF Tete(L)=e THEN Rechercher:=1
     ELSE Rechercher:=1+Rechercher(e,
          Queue(L))
 ELSE Rechercher:= -1
END;
```

4. Une version récursive :

```
FUNCTION Extraire(d,f:WORD; L:Liste):
     Liste;
BEGIN
 IF d>f THEN Extraire:=NIL
 ELSE Extraire:=Colle_en_tete(ieme(L,d)
     ,Extraire(d+1,f,L))
END;
```

5. En utilisant les fonctions précédentes :

```
FUNCTION InsererMilieu(e:Element;pos:WORD
     ;L:Liste):Liste;
BEGIN
 IF pos=1 THEN InsererMilieu:=
     Colle_en_tete(e,L)
 ELSE
 InsererMilieu:=Concat(Concat(Extraire(1,
     pos-1,L),Colle_en_tete(e,NIL)),
     Extraire(pos,longueur(L),L))
END;
```

6. Pour le test `.>3` :

```
FUNCTION Filtre(L:Liste):Liste;
BEGIN
 IF Est_Vide(L) THEN Filtre:=L
 ELSE
   IF Tete(L) > 3 THEN
         Filtre:=Colle_en_tete(Tete(L),
             Filtre(Queue(L)))
   ELSE
         Filtre:=Filtre(Queue(L))
END;
```

7. Toujours récursivement :

```
FUNCTION Supprimer(pos:WORD ; L:Liste):
     Liste;
BEGIN
 IF (pos >=1) AND (pos <= Longueur(L))
     THEN
  IF pos=1 THEN Supprimer:=Queue(L)
  ELSE Supprimer:=Colle_en_tete(Tete(L),
      Supprimer(pos-1,Queue(L)))
 ELSE Supprimer:=L
END;
```

Toutes ces fonctions peuvent être testées sous Pascal avec le programme principal suivant :

```
VAR L1,L2,L:Liste;
    i,d,f,pos1,poss : INTEGER;
    e,e1 : Element;

BEGIN
 L1:=Cree;
```

```
Affiche(L1);
WRITELN;
WRITELN(' La longueur de la liste est : ',
    Longueur(L1));
L2:=Cree;
Affiche(L2);
WRITELN;
WRITELN(' La longueur de la liste est : ',
    Longueur(L2));
L:=Concat(L1,L2);
WRITELN(' La liste concatenee est : ');
Affiche(L);
WRITELN;
WRITELN(' La longueur de la liste concatenee
    est : ',Longueur(L));
WRITELN(' Entrez un entier positif ');
READLN(i);
IF i<= Longueur(Concat(L1,L2)) THEN
   WRITELN('Le ',i,'eme element de la liste
        concatenee est : ',ieme(L,i))
ELSE
   WRITELN('Cet entier est trop grand');
WRITELN;
WRITELN(' La liste 1 a l''envers est : ');
Affiche(Retourne(L1));
 WRITELN(' La liste 2 a l''envers est : ');
Affiche(Retourne2(L2));
WRITELN;
WRITELN(' Entrer un element  :');
READLN(e);
IF Appartient(e,L) THEN
   BEGIN
   WRITELN(e,' appartient a la liste.');
   WRITELN('Il apparait pour la premiere fois
         au rang ',Rechercher(e,L))
   END
ELSE
   BEGIN
   WRITELN(e,' n''appartient pas a la liste 1
        .');
   END;
WRITELN;
WRITELN(' Entrer le rang du premier element
    de la liste extraite de L1 : ');
READLN(d);
WRITELN(' Entrer le rang du dernier element
    de la liste extraite de L1 : ');
READLN(f);
WRITELN(' La liste extraite est : ');
Affiche(Extraire(d,f,L1));
WRITELN(' Entrer l''element a inserer :');
READLN(e1);
WRITELN('Entrer le rang d''insertion :');
READLN(pos1);
Affiche(InsererMilieu(e1,pos1,L));
WRITELN('La liste L filtree est :');
Affiche(Filtre(L));
WRITELN('Entrer le rang a supprimer :');
READLN(poss);
 Affiche(Supprimer(poss,L));
 WRITELN('Appuyer sur Entree pour terminer.');
 READLN;
END.
```

Ce qui donne :

```
Rentrer une valeur : 5
Rentrer une valeur : 2
Rentrer une valeur : 3
Rentrer une valeur : 4
Rentrer une valeur : 6
Rentrer une valeur : 1
Rentrer une valeur : -1
     5   2   3   4   6   1
La longueur de la liste est : 6

Rentrer une valeur : 8
Rentrer une valeur : 7
Rentrer une valeur : 9
Rentrer une valeur : -1
     8   7   9
La longueur de la liste est : 3

La liste concatenee est :
   5   2   3   4   6   1   8   7   9
La longueur de la liste concatenee est:9

Entrez un entier positif
5
Le 5eme element de la liste concatenee est:6

La liste 1 a l'envers est :
     1     6     4     3     2     5
La liste 2 a l'envers est :
     9     7     8

Entrer un element  :
7
7 appartient a la liste.
Il apparait pour la premiere fois au rang 8

Entrer le rang du premier element
de la liste extraite de L1 :
2
Entrer le rang du dernier element
de la liste extraite de L1 :
5
La liste extraite est :
     2     3     4     6

Entrer l'element a inserer :
77
Entrer le rang d'insertion :
3
5 2 77 3 4 6 1 8 7 9

La liste L filtree est :
     5     4     6     8     7     9
```

```
Entrer le rang a supprimer :
3
 5    2    4    6    1    8    7    9
```

3 - 2 Liste des diviseurs

```
    FUNCTION ListeDiv(n:LONGWORD):Liste;
VAR L:Liste;
    k,fin:LONGWORD;
BEGIN
    L:=NIL;
    fin:=n;
    k:=1;
    WHILE k<fin DO
     BEGIN
      IF n MOD k = 0 THEN
        BEGIN
          fin:=n DIV k;
          IF k=fin THEN L:=Colle_en_tete(k,L)
          ELSE
           BEGIN
            L:=Colle_en_tete(k,L);
            L:=Colle_en_queue(n DIV k,L)
           END;
        END;
      k:=k+1
     END;
    ListeDiv:=L
END;
```

3 - 3 Liste des nombres parfaits

On a besoin de quelques fonctions intermédiaires de plus et les résultats précédents :

```
FUNCTION Som_list(L:Liste):Element;
BEGIN
IF L=NIL THEN Som_list:=0
ELSE Som_list:=Tete(L)+Som_List(Queue(L))
END;

FUNCTION Parfait(n:Element):BOOLEAN;
BEGIN
 IF Som_list(ListeDiv(n))=2*n THEN Parfait:=
     TRUE
 ELSE Parfait:=FALSE
END;
```

Maintenant la liste elle-même :

```
FUNCTION Liste_parfait(n:Element):Liste;
BEGIN
IF n=1 THEN Liste_parfait:=NIL
ELSE IF Parfait(n) THEN
          Liste_parfait:=Colle_en_tete(n,
              Liste_parfait(n-1))
      ELSE Liste_parfait:=Liste_parfait(n-1)
END;
```

Par exemple :

```
Les nombres parfaits inferieurs
a 100000 sont :

 8128  496   28   6
```

3 - 4 Crible d'Ératosthène récursif

```
FUNCTION inter(a,b:Element):Liste;
BEGIN
IF a>b THEN inter:=NIL
ELSE inter:=Colle_en_tete(a,inter(a+1,b))
END;


FUNCTION retmultiples(L:Liste;n:Element):Liste
    ;
BEGIN
 IF L=NIL THEN retmultiples:=NIL
 ELSE IF Tete(L) MOD n = 0 THEN
        retmultiples:=retmultiples(Queue(L),n
            )
      ELSE
        retmultiples:=Colle_en_tete(Tete(L),
            retmultiples(Queue(L),n))
END;


FUNCTION crible(m:Element):Liste;
  FUNCTION crible_temp(L:Liste;n:Element):
      Liste;
    BEGIN
     IF n>m THEN crible_temp:=NIL
     ELSE IF n*n>m THEN crible_temp:=
         Colle_en_tete(n,crible_temp(L,n+1))
          ELSE crible_temp:=Colle_en_tete(n,
              retmultiples(crible_temp(L,n+1),
              n))
  END;
BEGIN
  crible:=crible_temp(inter(2,m),2)
END;
```

3 - 5 Décomposition en facteurs premiers récursive

```
FUNCTION crible(m:Element):Liste;
  FUNCTION crible_temp(L:Liste;n:Element):
      Liste;
    BEGIN
     IF n>m THEN crible_temp:=NIL
     ELSE IF n*n>m THEN crible_temp:=
         Colle_en_tete(n,crible_temp(L,n+1))
```

```
        ELSE crible_temp:=Colle_en_tete(n,
             retmultiples(crible_temp(L,n+1),
             n))
  END;
  BEGIN
  crible:=crible_temp(inter(2,m),2)
END;
```

3 - 6 Changement de base récursif

```
FUNCTION base(n:Element;b:WORD):Liste;
BEGIN
 IF n<b THEN base:=Colle_en_tete(n,NIL)
 ELSE base:=Colle_en_queue(n MOD b,base(n DIV
      b,b))
END;
```

3 - 7 X 2004

1. On montre par récurrence que le coût en nombre d'appels récursifs est $n+1$:

```
FUNCTION card(X:ensemble):INTEGER;
BEGIN
IF X=NIL THEN card:=0
ELSE card:=1+card(X^.suivant)
END;
```

2. C'est le même type d'appels donc le même coût :

```
FUNCTION nPetits(p:INTEGER; X:ensemble):
     INTEGER;
BEGIN
IF X=NIL THEN nPetits:=0
ELSE IF X^.contenu<p THEN nPetits:=1+
     nPetits(p,X^.suivant)
      ELSE nPetits:=nPetits(p,X^.suivant)
END;
```

3. Encore les mêmes complexités :

```
FUNCTION partitionP(p:INTEGER; X:ensemble
     ):ensemble;
BEGIN
IF X=NIL THEN partitionP:=NIL
ELSE IF X^.contenu<p THEN
      partitionP:=cons(X^.contenu,
          partitionP(p,X^.suivant))
      ELSE partitionP:=partitionP(p,X^.
          suivant)
END;
```

```
FUNCTION partitionG(p:INTEGER; X:ensemble
     ):ensemble;
BEGIN
IF X=NIL THEN partitionG:=NIL
ELSE IF X^.contenu>p THEN
      partitionG:=cons(X^.contenu,
          partitionG(p,X^.suivant))
      ELSE partitionG:=partitionG(p,X^.
          suivant)
END;
```

4. On introduit des variables pour plus de lisibilité :

```
FUNCTION elementDeRang(k:INTEGER; X:
     ensemble):INTEGER;
VAR c:INTEGER;
    P,G:ensemble;
BEGIN
P:= partitionP(X^.contenu,X^.suivant);
G:= partitionG(X^.contenu,X^.suivant);
c:= card (P);
IF k <= c THEN
 elementDeRang:=elementDeRang(k,P)
ELSE IF k = c+1 THEN
      elementDeRang:=X^.contenu
     ELSE
      elementDeRang:=elementDeRang(k-(c
          +1),G)
END;
```

5. Dans le cas le plus défavorable, il peut y avoir $n+1$ appels récursifs. À chaque appel, on appelle `partitionP` qui a une complexité de $c+1$ et `partitionG` qui a une complexité de $n-(c+1)$, et `card` dont la complexité est de $c+1$. Dans le pire des cas, c décroît de $n-1$ à 0. Dans le pire des cas, la complexité peut donc être quadratique.

6. Comme T est croissante, pour déterminer $\max\{T(i-1),T(n-i)\}$, on va poser $n=2p$ ou $n=2p+1$ selon la parité de n.

   Considérons par exemple le premier cas, l'autre se réglant de manière analogue. Alors pour $1 \leqslant i \leqslant p$,

$$\max\{T(i-1),T(n-i)\} = T(n-i)$$

et après

$$\max\{T(i-1),T(i-1)\} = T(i-1)$$

Or

$$\begin{aligned}\sum_{i=1}^{p} \mathrm{T}(2p-i) &= \mathrm{T}(2p-1)+\cdots+\mathrm{T}(2p-p)\\ &= \mathrm{T}(p+1-1)+\cdots+\mathrm{T}(2p-1)\end{aligned}$$

donc

$$\sum_{i=1}^{2p} \max\{\mathrm{T}(i-1),\mathrm{T}(n-i)\} = 2\sum_{i=p+1}^{2p} \mathrm{T}(i-1)$$

On n'a pas d'idée a priori pour c alors essayons de voir ce que peux donner l'hérédité en gardant un c inconnu et en effectuant une récurrence forte.

$$\begin{aligned}\mathrm{T}(n) &\leqslant \ell n+\frac{2}{n}c\sum_{i=p+1}^{2p} i-1\\ &\leqslant \ell n+\frac{2}{n}c\sum_{i=p}^{2p-1} i\\ &\leqslant \ell n+\frac{2}{n}c\left(\frac{2p(2p-1)}{2}-\frac{p(p-1)}{2}\right)\\ &\leqslant \ell n+\frac{2}{2p}c\frac{3p^2-p}{2}\\ &\leqslant \ell n+\frac{2}{2p}c\frac{3p^2-p}{2}\\ &\leqslant \ell n+c\frac{3p-1}{2}\\ &\leqslant \ell n+c\frac{6p}{4}\\ &\leqslant \ell n+c\frac{3n}{4}\\ &\leqslant \left(\ell+\frac{3c}{4}\right)\end{aligned}$$

On voudrait donc que $\ell+\frac{3c}{4}=c$ i.e. $c=4\ell$. Sur sa copie on peut donc rédiger la récurrence forte avec $c=4\ell$.

7. On introduit des fonctions intermédiaires pour prendre des paquets de 5 éléments et pour créer les ensembles restant :

```
FUNCTION npremiers(X:ensemble;n:INTEGER):
    ensemble;
BEGIN
IF n=0 THEN npremiers:=NIL
ELSE npremiers:=cons(X^.contenu,npremiers
    (X^.suivant,n-1))
END;
```

```
FUNCTION nqueue(X:ensemble;n:integer):
    ensemble;
BEGIN
 IF n=1 THEN nqueue:=X^.suivant
 ELSE nqueue:=nqueue(X^.suivant,n-1)
END;

FUNCTION medians(X:ensemble):ensemble;
BEGIN
IF card(X)<5 THEN medians:=NIL
ELSE medians:=cons(elementDeRang(3,
    npremiers(X,5)),medians(nqueue(X,5)))
END;
```

8. On utilise `elementDeRangBis` pour déterminer le médian de `Y` :

```
FUNCTION elementDeRangBis(k:INTEGER;X:
    ensemble):INTEGER;
VAR cy,my,cp:INTEGER;
    P,G,Y:ensemble;
BEGIN
 Y:=medians(X);
 cy:=card(Y);
 IF cy<1 THEN elementDeRangBis:=
     elementDeRang(k,X)
 ELSE
  BEGIN
  my:=elementDeRangBis(((cy+1) DIV 2),Y);
  P:=partitionP(my,X);
  G:=partitionG(my,X);
  cp:=card(P);
  IF k <= cp THEN
   elementDeRangBis:=elementDeRangBis(k,P
       )
  ELSE IF k = cp+1 THEN
        elementDeRangBis:=my
       ELSE
        elementDeRangBis:=
            elementDeRangBis(k-cp-1,G)
  END;
END;
```

9. Le ℓ' introduit nous incite à rechercher les appels dont le coût est au maximum n. Il s'agit de :

- `Y:=medians(X);`
- `cy:=card(Y);`
- `P:=partitionP(my,X);`
- `G:=partitionG(my,X);`
- `cp:=card(P);`

Tout ceci se retrouve dans un $\ell' n$. Il reste à s'occuper des appels récursifs.

Le premier, $\mathtt{m}_{\mathrm{Y}}$, est en $\mathrm{M}'(c_{\mathrm{Y}})$. Or le cardinal c_{Y} de Y est $\left\lfloor \frac{n}{5} \right\rfloor$ donc son coût est $\mathrm{M}'\left(\left\lfloor \frac{n}{5} \right\rfloor\right)$.
Il reste les deux appels sur P ou G. On peut supposer que M′ est croissante comme T dans la question 5. Il s'agit donc maintenant de s'occuper de la taille de P et de celle de G or celles-ci sont liées donc on va majorer la taille de P.
Comme m_{Y} est le médian de Y, il y a $\left\lfloor \frac{\lfloor \frac{n}{5} \rfloor - 1}{2} \right\rfloor$ éléments de Y strictement inférieurs à m_{Y} et $\left\lfloor \frac{\lfloor \frac{n}{5} \rfloor + 1}{2} \right\rfloor$ éléments de Y strictement supérieurs à m_{Y}.
Maintenant, un élément donné e_{Y} de Y est le médian de 5 éléments de X. On s'occupe des éléments de X strictement inférieurs à m_{Y}.
Soit $e_{\mathrm{Y}} < m_{\mathrm{Y}}$, alors au plus cinq éléments de X sont strictement inférieurs à m_{Y}. Si $e_{\mathrm{Y}} \geqslant m_{\mathrm{Y}}$, au maximum 2 éléments de X sont strictement inférieurs à m_{Y}.
Il peut y avoir au plus quatre éléments de X qui ne sont pas pris en compte pour la détermination des médians des groupes de 5.
Finalement :

$$\begin{aligned} \mathrm{Card}\,\mathrm{P} &\leqslant 5\left\lfloor \frac{\lfloor \frac{n}{5} \rfloor - 1}{2} \right\rfloor + 2\left\lfloor \frac{\lfloor \frac{n}{5} \rfloor + 1}{2} \right\rfloor + 4 \\ &\leqslant 7\left\lfloor \frac{n}{10} \right\rfloor + 4 \end{aligned}$$

Donc :

$$\mathrm{M}'(n) \leqslant \ell' n + \mathrm{M}'\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + \mathrm{M}'\left(7\left\lfloor \frac{n}{10} \right\rfloor + 4\right)$$

Pour c', on s'inspire de ce qui a été fait à la question 6.
Au brouillon, on cherche un c' qui pourrait convenir :

$$\begin{aligned} \mathrm{M}'(n) &\leqslant \ell' n + \mathrm{M}'\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + \mathrm{M}'\left(7\left\lfloor \frac{n}{10} \right\rfloor + 4\right) \\ \mathrm{M}'(n) &\leqslant \ell' n + c' \cdot \left\lfloor \frac{n}{5} \right\rfloor + c' \cdot \left(7\left\lfloor \frac{n}{10} \right\rfloor + 4\right) \\ \mathrm{M}'(n) &\leqslant \left(\ell' + \frac{9}{10}c'\right) n + 4c' \end{aligned}$$

Il existe n_k tel que, pour tout $n \geqslant n_k$, $4 \leqslant \frac{n}{k}$ pour un certain k qu'on cherche à déterminer.

Alors

$$\mathrm{M}'(n) \leqslant \left(\ell' + \left(\frac{9}{10} + \frac{1}{k}\right) c'\right) n$$

On cherche donc k pour que c' vérifie :

$$c' = \ell' + \left(\frac{9}{10} + \frac{1}{k}\right) c'$$

c'est-à-dire :

$$c'\left(\frac{1}{10} - \frac{1}{k}\right) = \ell'$$

Il suffit donc que le coefficient de c' soit positif. On prend par exemple $k = 20$, alors

$$c' \geqslant 20\ell'$$

Il faut également que $\mathrm{M}'(n) \leqslant c'n$ pour $n \leqslant n_k$.
On montre alors sur sa copie le résultat à l'aide d'une récurrence forte.

10. On reprend le travail effectué précédemment en adaptant, notamment pour le calcul de Card P :

$$\begin{aligned} \mathrm{Card}\,\mathrm{P} &\leqslant 3\left\lfloor \frac{\lfloor \frac{n}{3} \rfloor - 1}{2} \right\rfloor + \left\lfloor \frac{\lfloor \frac{n}{3} \rfloor + 1}{2} \right\rfloor + 2 \\ &\leqslant 4\left\lfloor \frac{n}{6} \right\rfloor + 2 \end{aligned}$$

ce qui donne finalement :

$$\mathrm{M}'(n) \leqslant \ell' n + \mathrm{M}'\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + \mathrm{M}'\left(4\left\lfloor \frac{n}{6} \right\rfloor + 2\right)$$

Si nous continuons , des problèmes surviennent :

$$\begin{aligned} \mathrm{M}'(n) &\leqslant \ell' n + \mathrm{M}'\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + \mathrm{M}'\left(4\left\lfloor \frac{n}{6} \right\rfloor + 2\right) \\ \mathrm{M}'(n) &\leqslant \ell' n + c' \cdot \left\lfloor \frac{n}{3} \right\rfloor + c' \cdot \left(2\left\lfloor \frac{n}{3} \right\rfloor + 2\right) \\ \mathrm{M}'(n) &\leqslant \left(\ell' + c'\right) n + 2c' \end{aligned}$$

Il existe n_k tel que, pour tout $n \geqslant n_k$, $2 \leqslant \frac{n}{k}$ pour un certain k qu'on cherche à déterminer.

Alors
$$\mathrm{M}'(n) \leqslant \left(\ell' + \left(1+\frac{1}{k}\right)c'\right)n$$

On cherche donc k pour que c' vérifie :

$$c' = \ell' + \left(1+\frac{1}{k}\right)c'$$

Aïe, ça ne marche plus ! Ça ne prouve pas pour autant que ça ne peut pas marcher autrement...Mais comme on demande de prouver que le coût n'est plus au pire linéaire, il faudrait minorer $\mathrm{M}'(n)$ par quelque chose de plus coûteux.

Comme nous avons une majoration de $\mathrm{M}'(n)$, ce n'est pas pratique. Cependant, en réorganisant X, on peut s'arranger pour qu'il y ait égalité pour chaque n en remplaçant les « au maximum » par des égalités dans notre raisonnement.

Alors, pour se débarrasser des parties entières, on peut considérer, pour tout naturel k, des entiers $n \geqslant 3^k$.

Alors

$$\begin{aligned}\mathrm{M}'(n) &\geqslant \ell' n + \mathrm{M}'\left(\frac{3^k}{3}\right) + \mathrm{M}'\left(4\frac{3^k}{6}+2\right)\\ &\geqslant \ell' n + \mathrm{M}'\left(3^{k-1}\right) + \mathrm{M}'\left(2\cdot 3^{k-1}\right)\end{aligned}$$

On peut alors montrer par récurrence que si $n \geqslant 3^k$, on a $\mathrm{M}'(n) \geqslant k\ell' n$.

En effet, pour l'hérédité, on peut appliquer l'hypothèse de récurrence à la dernière inégalité obtenue :

$$\begin{aligned}\mathrm{M}'(n) &\geqslant \ell' n + \mathrm{M}'\left(3^{k-1}\right) + \mathrm{M}'\left(3^{k-1}\right)\\ \mathrm{M}'(n) &\geqslant \ell' n + (k-1)\ell' n + (k-1)\ell' n\\ \mathrm{M}'(n) &\geqslant \ell' n(2k-1)\\ \mathrm{M}'(n) &\geqslant \ell' nk\end{aligned}$$

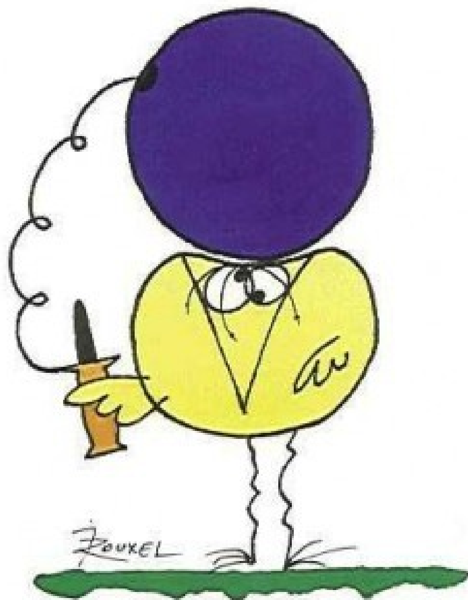
On ne peut donc pas trouver un majorant de la forme $c'n$ avec c' indépendant de n.

Guillaume Connan, Licence Creative Commons BY: $\circledS$ $\circledcirc$, 2010-2011

OPTION

4

Logique des propositions



1 Généralités

On appellera **proposition** tout énoncé dont on peut décider s'il est vrai ou faux indépendamment de ses composantes.
On distinguera la logique des propositions de la logique des **prédicats** qui introduit des variables dans les assertions qui peut les rendre donc parfois vraies et parfois fausses et que nous étudierons en deuxième année.
Par exemple « *Igor dort* » est une proposition susceptible d'être vraie ou fausse dans toute situation alors que la valeur de vérité de « *x dort* » dépend de x.
Nous distinguerons également la **syntaxe** de la logique des propositions de sa SÉMANTIQUE.
Le dictionnaire propose les définitions suivantes :

- **syntaxe** : n. f. (bas latin syntaxis, du grec suntaksis, ordre) Partie de la grammaire qui décrit les règles par lesquelles les unités linguistiques se combinent en phrases.
- **sémantique** : n. f. (bas latin semanticus, du grec sêmantikos, qui signifie) **1.** Étude du sens des unités linguistiques et de leurs combinaisons. **2.** Étude des propositions d'une théorie déductive du point de vue de leur vérité ou de leur fausseté.

L'étude de l'aspect syntaxique consiste à préciser comment l'on construit les formules et l'aspect sémantique interprète les formules en terme de *Vrai* ou *Faux*.

2 Syntaxe

2 1 Les symboles

L'alphabet du langage de la logique des propositions est constitué de :

- un ensemble fini ou infini dénombrable de *variables propositionnelles* ;
- la constante logique $\perp$ qui se lit « FAUX » ;
- les *connecteurs* $\neg$ (non), $\wedge$ (et), $\vee$ (ou), $\rightarrow$ (implique) et $\leftrightarrow$ (équivalent) ;
- les parenthèses « (» et «) ».

Le connecteur $\neg$ est un *connecteur unaire* et les autres sont des *connecteurs binaires*.
L'*ensemble des formules* de la logique propositionnelle est le plus petit ensemble $\mathscr{C}$F tel que :

- toute variable propositionnelle est un élément de $\mathscr{F}$;
- $\perp$ est un élément de $\mathscr{F}$;
- si $p \in \mathscr{F}$, alors $(\neg p) \in \mathscr{F}$;
- si p et q sont dans $\mathscr{F}$, alors $(p \wedge q)$, $(p \vee q)$, $(p \rightarrow q)$, et $(p \leftrightarrow q)$ sont des éléments de $\mathscr{F}$.

Guillaume Connan, Licence Creative Commons BY: $\circledS$ $\circledC$, 2010-2011

Par exemple, $s \to ((p \vee (\neg q)) \vee (p \vee (\neg r)))$ est une proposition mais $\neg \wedge (p) \vee$ ne l'est pas.

On peut se passer de quelques parenthèses en adoptant les règles de priorité suivantes :

- $\neg$ est prioritaire sur les autres opérateurs ;
- $\vee$ et $\wedge$ sont prioritaires sur $\to$ et $\leftrightarrow$.

Mais attention ! $p \vee q \wedge r$ est ambigu.

2 2 Démonstration par induction

Théorème 4 - 1

Si une propriété P portant sur les formules de $\mathscr{F}$ est telle que :

- toute variable propositionnelle vérifie P ;
- $\perp$ vérifie P ;
- si la formule p vérifie P, alors $(\neg p)$ vérifie P ;
- si p et q vérifient P, alors $(p \vee q)$, $(p \wedge q)$, $(p \to q)$ et $(p \leftrightarrow q)$ vérifient P ;

alors toutes les formules de $\mathscr{F}$ vérifient P.

Ce théorème permet par exemple de prouver le théorème suivant :

Théorème 4 - 2

Toute formule de $\mathscr{F}$ a autant de parenthèses ouvrantes que de parenthèses fermantes.

La preuve est laissée en exercice.

2 3 Sous-formules

Nous n'entrerons pas dans les détails mais on peut définir par induction les sous-formules d'une formule donnée : on retiendra juste qu'il s'agit de toutes les formules apparaissant dans une formule donnée.

Par exemple, l'ensemble des sous-formules de $(p \to q) \vee \neg(q \leftrightarrow r)$ est

$$\{p \to q, \neg(q \leftrightarrow r), q \leftrightarrow r, p, q, r\}$$

3 Sémantique

3 1 Valeurs et tables de vérité

On considère un ensemble $\mathscr{B} = \{0,1\}$ ou $\{F,V\}$ ou $\{\text{Oui},\text{Non}\}$, etc.

Définition 4 - 1

Une *distribution des valeurs de vérité* est une application v de $\mathscr{F}$ dans $\mathscr{B}$ définie inductivement par :

- $v(\perp) = 0$,
- $v(\neg p) = 1$ si, et seulement si, $v(p) = 0$;
- $v(p \wedge q) = 1$ si, et seulement si, $(p) = v(q) = 1$;
- $v(p \vee q) = 0$ si, et seulement si, $v(p) = v(q) = 0$;
- $v(p \to q) = 0$ si, et seulement si, $v(p) = 1$ et $v(q) = 0$;
- $v(p \leftrightarrow) = 1$ si, et seulement si, $v(p) = v(q)$,

avec p et q des éléments quelconques de $\mathscr{F}$.

Cette définition permet d'établir les *tables de vérité* suivantes :
puis de généraliser à n'importe quelle formule :

3 2 Calcul propositionnel et Calcul dans F$_2$

On muni $\{0,1\}$ des opérations $+$ et $\cdot$. On n'oubliera pas que dans $\mathbb{F}_2$, $x + x = 0$ et $x^2 = x$.
On peut alors définir une distribution des valeurs de vérité cette fois à valeurs dans $\mathbb{F}_2$:

- $v(\perp) = 0$;
- $v(p) = x \in \{0,1\}$; $v(q) = y \in \{0,1\}$
- $v(\neg p) = 1 + x$;
- $v(a \vee q) = x + y + x \cdot y$;
- $v(p \wedge q) = x \cdot y$;
- $v(p \to q) = 1 + x + x \cdot y$;
- $v(p \leftrightarrow q) = 1 + x + y$.

Guillaume Connan, Licence Creative Commons BY: $\textcircled{\$}$ $\textcircled{=}$, 2010-2011